O'REILLY®

TURING 图灵程序设计丛书

第4版



C#经典实例

C# 6.0 Cookbook

针对C# 6.0和.NET Framework 4.6全面更新；涵盖C#开发的各类陷阱和问题

[美] Jay Hilyard　Stephen Teilhet 著
徐敬德 译

中国工信出版集团　人民邮电出版社 POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

C#经典实例（第4版）

C# 6.0 Cookbook

[美] Jay Hilyard Stephen Teilhet 著
徐敬德 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　京

图书在版编目（CIP）数据

C#经典实例 ：第4版 / (美) 杰伊·希尔亚德
(Jay Hilyard) , (美) 斯蒂芬·泰耶
(Stephen Teilhet) 著 ; 徐敬德译. -- 北京 : 人民邮
电出版社, 2016.10
（图灵程序设计丛书）
ISBN 978-7-115-43509-5

Ⅰ. ①C… Ⅱ. ①杰… ②斯… ③徐… Ⅲ. ①C语言一
程序设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2016)第216681号

内 容 提 要

本书共分为13章，每一章侧重于特定主题的C#解决方案。具体内容包括：类和泛型，集合、枚举器和迭代器，数据类型，语言集成查询和lambda表达式，调试和异常处理，反射和动态编程，正则表达式，文件系统I/O，网络和Web，XML，安全，线程、同步和并发，工具箱。本书使用大量范例，帮助开发人员快速理解并解决现实中的问题。

本书面向所有级别的C#和.NET开发人员。

◆ 著　　　[美] Jay Hilyard　Stephen Teilhet
译　　　徐敬德
责任编辑　朱　巍
执行编辑　杨　琳　赵瑞琳
责任印制　彭志环

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：37
字数：874千字　　　　2016年10月第1版
印数：1－3 500册　　　2016年10月北京第1次印刷

著作权合同登记号　图字：01-2016-4798号

定价：129.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第8052号

O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

前言

C# 是一门面向 Microsoft .NET 平台开发者的语言。Microsoft 将 C# 描述为一种用于 .NET 平台开发的富于创新的现代化语言，并且在 C# 6.0 中增添了用于支持动态编程、并行编程以及编写更少代码的新功能，进一步实现了这个目标。C# 不仅同时支持声明式编程和函数式编程，还包含了强大的面向对象特性。简而言之，用 C# 可以针对特定问题采取不同的编程风格。

我们基于自己最初学习 C# 时遇到的编程问题开始了本书的编写，并且根据 C# 语言中的新问题和新功能不断地扩展内容。在这一版中，我们重新编写了许多解决方案，以充分利用 C# 最近的创新，例如新的表达式级别功能（`nameof`、字符串插值、`null` 条件运算符、索引初始值设定项）、成员声明功能（自动属性初始值设定项、getter-only 自动属性、表达式 – 函数体成员）和语句级别功能（异常过滤器）。同时，我们在原有及新的范例中纳入了动态编程（C# 4.0）和异步编程（C# 5.0）的新应用，帮助读者了解如何应用这些语言特性。

无论是首次学习 C#，还是探索其新的能力，抑或是处理开发周期中较为罕见的问题，每个人都会遇到一些常见（和不那么常见的）陷阱和问题；希望我们的以上增补能帮助读者解决难题。此外，尽管 Microsoft 已经提供了大量的功能以避免人们“重复创造轮子”，但是我们仍然发现 .NET Framework 类库（Framework class library，FCL）中有一些缺少的内容，并将其纳入了本书的范例。一些解决方案你可能马上会用到，也有一些你可能永远都用不到，但是无论如何，我们都希望这本书能够帮助你尽可能充分地利用 C# 和 .NET Framework。

本书的内容按照一个 C# 程序员在学习过程中要解决的问题类型来进行组织。这些解决方案称为*范例*（recipe）；每个范例都包含一个问题、其解决方案、对解决方案的讨论和其他相关信息，最后是一个资源列表，包括在 FCL 的哪里可以找到相关类的更多信息、相关文章和其他范例。这种问答模式提供了问题的完整解决办法，使得本书易于阅读和使用。几乎每个范例都包含完整的、有文档的代码示例，向读者展示如何解决特定的问题，同时也讨论了底层技术如何运作，并且列出了替代技术、限制条件和应用时要考虑的其他事项。

读者对象

即使你不是经验丰富的 C# 或 .NET 开发人员，也可以使用本书——本书面向所有级别的读

者。本书既提供了开发人员日常问题的解决方案，也包含了一些出现频率较低的问题。书中范例针对的是现实开发人员需要立刻解决的问题，不需要首先学习大量理论知识。虽然参考书和教程类书籍可以讲述通用概念，但通常不提供读者解决实际问题所需的帮助。我们选择通过示例来教学，因为这是大多数人自然的学习方式。

本书中解决的大多数问题是 C# 开发人员会频繁面对的，但有一些更高级的问题需要结合多项技术的复杂方案。每个范例旨在帮助开发人员快速理解问题，学习如何解决，并找出任何潜在的权衡选择来帮助你快速、高效、轻松地解决问题。

为了免去读者手动输入解决方案的麻烦，我们在 O'Reilly 的网站上提供了本书的示例代码，以方便"编辑继承"模式的开发（复制和粘贴），同时有助于经验较少的开发人员看到优秀的编程实践。示例代码提供了利用到每个解决方案的可运行测试，不过本书也在每个解决方案中包含了足够的代码，读者不使用示例代码也能够实现解决方案。示例代码可以从本书的产品页面（https://github.com/oreillymedia/c_sharp_6_cookbook）上获取[1]。

硬件和软件要求

要运行本书中的示例，你需要一台运行 Windows 7 或更高版本的计算机。一部分网络和 XML 解决方案需要 Microsoft IIS 7.5 或更高版本，第 9 章中 FTP 的范例需要一个本地配置好的 FTP 服务器。

打开和编译本书中的示例需要 Visual Studio 2015。如果你精通可下载的 Framework SDK 及其命令行编译器，也可以顺利地使用本书和示例代码。

平台说明

本书中的解决方案都是使用 Visual Studio 2015 开发的。C# 6.0 和 C# 3.0 之间的差异是非常显著的，本书的示例代码与第 3 版中有很大不同，反映了其中的差异。

值得一提的是，尽管 C# 现在已经是 6.0 版，.NET Framework 则仍是 4.6 版。C# 随着 .NET Framework 的每次发布持续创新，现在的 C# 6.0 中包含许多功能，使得开发人员可以用最适合手头任务的风格进行编程。

内容结构

本书共分为 13 章，每一章侧重于特定主题的 C# 解决方案。下面总结了每一章的要点，概述了本书的内容。

- 第 1 章　类和泛型

  这一章篇幅很长，包含处理类和结构数据类型的范例，以及泛型的使用。泛型能够让你的代码在不同类型的值上运行一致。这一章涵盖的范例范围很广，包括闭包、类转换、完善的命令行参数处理系统以及类设计的主题。有的范例增强了读者对泛型的整

注 1：示例代码也可以在图灵社区本书主页（http://www.ituring.com.cn/book/1746）下载。——编者注

体理解；有的范例涵盖了泛型何时适用，框架中提供了哪些支持，以及如何实现自定义集合。

- 第 2 章　集合、枚举器和迭代器

  这一章范例考察了集合、枚举器和迭代器的用法。集合的范例使用了数组（单维、多维和锯齿状）、`List<T>` 以及其他集合类，并且扩展了它们的功能。这一章讨论了泛型集合以及创建自定义强类型集合的多种方式。我们探讨了自定义枚举器的创建，向读者展示了如何为泛型和非泛型类型实现迭代器以及如何使用迭代器实现 `foreach` 功能，并涵盖了自定义迭代器实现。

- 第 3 章　数据类型

  这一章包括字符串、数字和枚举。这些范例展示了如何完成诸如编码 / 解码字符串、执行数值转换，以及测试字符串以确定它们是否包含数字值的任务。我们还介绍了如何显示、转换和测试枚举类型，以及如何使用包含位标志的枚举。

- 第 4 章　语言集成查询和 lambda 表达式

  这一章涵盖语言集成查询（language integrated query，LINQ）及其用法，包括并行 LINQ（parallel LINQ，PLINQ）的一个示例。有些范例使用了诸多标准查询运算符，也有些展示了如何使用功能强大但并不是语言关键字的查询运算符。这一章也对 lambda 表达式进行了探讨，通过范例展示如何用其代替旧风格的委托。

- 第 5 章　调试和异常处理

  这一章介绍调试和异常处理。我们提供了使用 `System.Diagnostics` 命名空间下数据类型的范例，例如事件日志、进程、性能计数器和类型的自定义调试器显示。我们同时关注了在应用程序中实现异常处理的最佳方式。这一章还包括了避免未处理异常，读取和显示栈跟踪，引发和重新引发异常的范例。最后，我们提供的范例展示了如何克服某些棘手的情形，如后期绑定调用的方法中出现的异常和异步异常处理。

- 第 6 章　反射和动态编程

  这一章展示了如何使用 .NET Framework 内置的程序集检视系统，确定一个程序集中实现了哪些类型、接口和方法，以及如何以后期绑定的方式访问它们。这一章也说明了如何使用 `dynamic`、`ExpandoObject` 和 `DynamicObject` 在应用程序中实现动态编程。

- 第 7 章　正则表达式

  这一章介绍了一组有用的类，用于对字符串运行正则表达式。范例包括列举正则表达式匹配，将字符串解析为一组标记，查找 / 替换字符，以及验证正则表达式的语法。此外我们还加入了一个范例，其中包含许多常见的正则表达式模式。

- 第 8 章　文件系统 I/O

  这一章涉及与文件系统交互的三种不同方式：典型的文件交互；基于目录或文件夹的交互；文件系统 I/O 的高级主题。

- *第 9 章　网络和 Web*

  这一章探讨了由 .NET Framework 提供的连接选项，以及如何以编程方式访问网络资源和 Web 上的内容。这一章中的范例包含了直接使用 TCP/IP，使用命名管道通信，构建自己的端口扫描程序，以编程方式确定网站配置，等等。

- *第 10 章　XML*

  如果你在使用 .NET，那么很可能需要在一定程度上处理 XML。在这一章中，我们将探讨 XML 的一些作用，以及如何使用 LINQ to XML、`XmlReader`/`XmlWriter` 类和 `XmlDocument` 类来进行 XML 编程。这一章包含了使用 XPath 和 XSLT 的示例，以及验证 XML、将 XML 转换到 HTML 等主题。

- *第 11 章　安全*

  编写不安全代码的方式有很多种，但仅有少数几个途径可以编写安全的代码。在这一章中，我们探讨了类型的访问控制、加密和解密、安全地存储数据、编程式安全和声明式安全等领域。

- *第 12 章　线程、同步和并发*

  这一章的主题是在 .NET 程序中使用多个执行线程，讨论的问题有：在应用中实现多线程，避免资源的并行访问，允许安全的并行访问，存储每个线程的数据，顺序执行任务，在 .NET 中使用同步原语以编写线程安全的代码，等等。

- *第 13 章　工具箱*

  这一章包含的范例是开发人员会反复遇到的随机操作类型，例如确定系统资源的位置，发送电子邮件，以及使用服务。这一章还包括一些较少用到但非常有用的应用程序块，如消息队列，在单独的应用程序域中运行代码，以及在*全局程序集缓存*（global assembly cache，GAC）中查找应用程序集的版本。

有一些范例是相关联的；在这类范例中，参阅部分和讨论部分的文本中会注明这些关联关系。

未涉及的内容

这本书并不是 C# 的参考手册或入门书。O'Reilly 出版了一些优秀的入门书和参考手册，如 Joseph Albahari 和 Ben Albahari 的 *C# 6.0 in a Nutshell*（http://shop.oreilly.com/product/0636920040323.do）和 *C# 6.0 Pocket Reference*（http://shop.oreilly.com/product/0636920040675.do），以及 Stephen Cleary 的《C# 并发编程经典实例》（*Concurrency in C# Cookbook*，http://shop.oreilly.com/product/0636920030171.do）。MSDN 库也是极为有用的。它包含在 Visual Studio 2015 中，也可以在网站 http://msdn.microsoft.com 上在线查看。

排版约定

在本书中将会使用以下排版约定。

- 等宽字体（`Constant width`）
  用于程序清单和代码元素，如命令、选项、开关、变量、特性、键、函数、类型、类、命名空间、方法、模块、属性、参数、值、对象、事件、事件处理器、XML 标记、HTML 标记、宏、文件的内容和命令输出。
- 加粗等宽字体（**`Constant width bold`**）
  用于在程序清单中突出显示代码中的重要部分。
- 斜体等宽字体（*`Constant width italic`*）
  用于指示代码中可替换的部分。
- `//...`
  C# 代码中的省略号表示为了段落清晰而省略掉的文本。
- `<!--...-->`
  在 XML 模式和文档的代码中的省略号表示为了段落清晰而省略掉的文本。

此标志指示提示、建议或一般注意事项。

此标志表示警告或警示。

关于代码

本书中几乎每个范例都包含一个或多个代码示例。这些示例包含在解决方案中，代码片段和完整项目都可以直接用于你的应用程序。大多数代码示例都写在一个类或结构中，使得它们更易于在应用程序中使用。除此之外，所有的 `using` 指令都包含在各个范例中，因此你不需要查明在你的代码中要包含哪个命名空间。

只有关键的部分才包括完整的错误处理，例如输入参数。这允许你轻松地查看什么是正确的输入，什么是错误的输入。许多范例省略了错误处理，关注重点概念会使得解决方案更易于理解。

使用代码示例

这本书的示例代码可从网页 https://github.com/oreillymedia/c_sharp_6_cookbook 上获取。

这本书是用来帮助你完成工作的。一般来说，你可以在自己的程序和文档中使用本书的代码。你没有必要联系我们来获得授权，除非想对代码做出大规模的重构。比如，使用本书中的若干程序来编写自己的代码是无需授权的，但是销售或分发 O'Reilly 出版书籍配套光盘中的代码是需要授权的；引用本书中的内容或示例代码来回答问题是无需授权的；而把本书中的大量代码合并到你的产品文档中就需要授权。

我们并不要求你注明引用内容的出处，但是非常感激你这么做。一条引用说明通常包括书名、作者、出版商和 ISBN。例如，"*C# 6.0 Cookbook*, Fourth Edition, by Jay Hilyard and Stephen Teilhet. Copyright ©2015 Jay Hilyard and Stephen Teilhet, 978-1-4919-2146-3"。

如果你感觉你使用代码示例的方式不属于以上所述的任何方式，可以随时与我们联系，我们的电子邮箱地址为 permissions@oreilly.com。

Safari®在线图书

Safari 在线图书（http://safaribooksonline.com/）是一个基于用户需求，发行全球技术和商业领域顶级作者的优质图书和视频（https://www.safaribooksonline.com/explore/）的数字图书馆。

技术专家、软件开发人员、网页设计师以及商业和创意专家都选择将 Safari 在线图书作为研究、解决问题、学习和证书培训的首要资源。

Safari 在线图书为企业（https://www.safaribooksonline.com/enterprise/）、政府（https://www.safaribooksonline.com/government/）、教育机构（https://www.safaribooksonline.com/academic-public-library/）和个人提供了不同的产品组合和价格方案（https://www.safaribooksonline.com/pricing/）。

用户可通过 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett 和 Course Technology 等数百个出版商的数据库（https://www.safaribooksonline.com/our-library/）搜寻上千种图书、培训视频和预出版的手稿。要了解有关 Safari 在线图书的更多信息，请访问我们的在线网站（http://safaribooksonline.com/）。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：
http://shop.oreilly.com/product/0636920037347.do

对于本书的评论和技术性问题，请发送电子邮件到：
bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：
http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

这本书始于我们最初接触 C#，并在我们多年来对该语言的探索和创新应用中演变成长。自本书的上一版出版之后，随着 C# 6.0 的发布以及 C# 4.0 和 C# 5.0 的新功能出现，我们决定复查前三个版本以确认如何改进原有的范例，并探索使用 C# 更好地完成编程任务的方法。通过对 C# 和 Framework 大量知识的不断学习，我们致力于在本书中为读者呈现 C# 是如何演变的，以及如何使用它更好地完成工作。

以下这些人对本书的完成具有不可或缺的作用，我们在此对他们给予的帮助表示感谢。

感谢 Brian MacDonald（我们的编辑）、Heather Scherer、Rachel Monaghan、Nick Adams 和 Sara Peyton。在你们的督促下本书才得以快速完成并出版。感谢你们的付出。

同时我们还要感谢我们的技术复审小组成员：Steve Munyan、Lee Coward 和 Nick Pinkham。感谢你们对完善本书的全身心投入和富有洞察力的见解。没有你们有价值的反馈，这本书就不可能存在，感谢你们。

Jay Hilyard的致谢

感谢 Steve Teihet 的创意、幽默感以及愿意再次与我踏上写书的征途。我一直很享受与你共事，尽管都是在晚上和周末，并且几乎都是在网络上。

感谢我的妻子 Brooke。尽管明白写书意味着我们相处时间的减少，但你依然支持我、鼓励我、帮助我。这本书没有你就不可能诞生。谢谢你，我爱你！

感谢我的两个儿子，Owen 和 Drew。你们从新角度看待事物的能力每每让我惊讶，我为你们所完成的一切感到骄傲。我很高兴你们都对我为之奋斗一生的领域感兴趣，你们是我无可替代的珍宝。

感谢 Phil 和 Gail 对我不得不牺牲假期而工作的支持和理解，并肩负起祖父母的责任帮助我照顾孩子，同时还要感谢妈妈每个月的心灵鸡汤。

感谢我的一群好友：Seth、Katie Fiermonti、Tom Bebbington 和 Jenna Roberts。世上再无难事，只要有朋友和一杯好酒。

感谢 Scott Cronshaw、Bill Bolevic、Melissa Jurkoic、Mike Kennie、Alex Shore、Dave Flanders、Aaron Reddish、Rakshit Jain、Jason Phelps、Josh Clairmont、Bob Blais、Kim Serpa、Stu Savage、Gaurang Patel、Jesse Peters、Ken Jones、Mahesh Unnikrishnan、T Antonio、Mary Ellen Sawyer、Jon Godbout、Atul Kaul、Mark Miller、Rich Labenski、Lance Simpson、Tim Beaulieu 和 Lee Horgan。你们是最棒的团队，你们工作都非常努力，我感谢你们所做的一切。

最后，再次感谢我的家人和朋友。你们对一本自己并不了解的书表示了极大的关心，并为我感到骄傲。

Steve Teilhet的致谢

我很骄傲能与 Jay Hilyard 成为好友，他是一个出色的合作者，一个勤奋的合著者。不是每天都能找到一个既值得信任又能合作无间的好朋友。再次与你合著是我的荣幸。

感谢我的妻子 Kandis Teilhet 给了我坚持前行每一步的力量。我对你的爱已尽在不言中。感谢我的两个儿子 Patrick 和 Nicholas，是你们让艰难的时刻变得顺利。你们对我来说无可替代。现在你们都步入了人生的下一个阶段，我为能看到你们即将获取的成就感到激动，没准你们也会写一本书。

感谢我的妈妈、爸爸和兄弟，感谢他们一直以来的倾听和支持。

最后同样要感谢的是 IBM 团队、Larry Rose、Babita Sharma、Jessica Berliner、Jeff Turnham、John Peyton、Kris Duer、Robert Stanzel、Shu Wang、Bingzhou Zheng、Dave Steinberg、Dave Stewart、Jason Todd、Alexei Pivkine、Joshua Clark、William Frontiero、Matthew Murphy、Omer Trip、Marco Pistoia、Enrique Varillas、Guillermo Hurtado、Bao Lu、Mary Santo、Diane Redfearn、Urmi Chatterjee、Joshua Ho、Kenneth Cheung、Andrew Mak、Daniel Nguyen、Jennifer Calder、Tahseen Shabab、Srinivas Sripada、David Marshak、Larry Gerard、Douglas Wilson、Steve Hikida 以及其他许多人，你们的辛勤工作和才华一直激励着我。

电子书

扫描如下二维码，即可购买本书电子版。

第 1 章

类和泛型

1.0 简介

本章的范例涵盖了 C# 语言的基础，主题包括类和结构，如何使用它们，它们有哪些不同，何时使用类以及何时使用结构。在此基础上，我们将构建具有各种固有功能（如可排序、可搜索、可处理和可克隆）的类。此外，我们将深入讨论联合类型、字段初始化、lambda、局部方法、单路和多路广播委托、闭包、函数对象等主题。本章也包含了解析命令行参数的范例，这是开发人员一直喜爱的主题。

在开始展示这些范例之前，让我们回顾一下关于类、结构、泛型的面向对象能力的关键信息。类比结构灵活得多。结构可以跟类一样实现接口，但与类不同的是，它们不能继承自类或结构。这种限制使得你无法创建结构层次关系，而这用类可以做到。通过抽象基类实现的多态性也是在结构中无法使用的，因为除了装箱成 `Object`、`ValueType` 和 `Enum`，结构无法从另一个类派生。

结构与其他的值类型一样，都是从 `System.ValueType` 隐式派生的。乍看之下，结构类似于类，但它们实际上有很大的差别。在设计应用程序时，知道何时使用结构优于使用类将对你有很大的帮助。不正确地使用结构可能会使代码性能低下、难以修改。

结构相对于引用类型有两个性能优势。首先，如果一个结构是在栈上分配的（即不包含在引用类型内），访问结构及其数据的速度要快于访问堆中引用类型的速度。

引用类型的对象必须要跟随它在堆上的引用以获取它们的数据。不过，这种性能优势相对于结构的第二个性能优势就相形见绌了：要清理在栈上为结构分配的内存，只需要在方法调用返回时修改栈指针所指向的地址即可。这个调用要远远快于垃圾回收器自动清理托管堆上分配的引用类型。然而垃圾回收器的成本是延后的，所以不会立刻被人注意到。

当以传值方式传入其他方法时，结构的性能就比不上类了。因为结构存在于栈上，当以传值方式传入一个方法时，结构及其数据必须复制到一个新的局部变量（方法用于接收结构的参数）中。这一复制过程相比将一个引用传入方法要花费更多的时间，除非结构的大小与机器的指针大小相同或更小一些；因此，在 32 位的机器上，传入一个 32 位大小的结构与传入一个引用（与指针大小相同）的成本是相同的。在类和结构之间选择时，要记得这一点。尽管创建、访问和销毁类对象可能需要更长时间，但并不能抵消将结构多次按值传入一个或多个方法产生的性能下降。保持较小的结构体可以减小按值传递时所产生的性能下降。

以下情况应使用类。

- 其同一性很重要。结构在按值传入方法时会被隐式复制。
- 有较大的内存占用。
- 其字段需要初始化。
- 需要从一个基类继承。
- 需要多态行为；也就是说，你需要实现一个抽象基类，并从此基类派生出多个相似的类。（注意，多态性也可以通过接口实现，但通常并不适合在一个值类型中实现接口。这是因为当结构转换为接口时，会因装箱操作而导致性能损失。）

以下情况应使用结构。

- 其行为方式类似于原语类型（`int`、`long`、`byte` 等）。
- 仅占用较小的内存。
- 调用一个需要将结构体以传值方式传入的 `P/Invoke` 方法。*平台调用*（Platform Invoke，P/Invoke）允许托管代码调用 DLL 内公开的非托管方法。许多时候，非托管 DLL 内的方法都需要传入一个结构参数。使用结构是执行此操作的一种高效方法，并且在需要按值传入时是唯一的途径。
- 需要降低垃圾回收对应用程序性能的影响。
- 其字段只需要被初始化为默认值。对于数值类型，这个值为 `0`；对于布尔类型，则为 `false`；对于引用类型，则为 `null`。注意在 C# 6.0 中，结构可以拥有默认构造函数并将字段初始化为非默认值。
- 不需要继承一个基类（除了 `ValueType` 之外，所有结构都继承它）。
- 不需要多态行为。

当把结构传递给需要一个对象参数的方法时，例如 Framework 类库（FCL）中的任何非泛型集合类型，它们也可能会引起性能降低。把一个结构（对此问题而言其实是任何简单类型）传入一个需要对象参数的方法中将会导致结构被装箱。*装箱*（boxing）是指将一个值类型包装在一个对象中。这种操作比较耗时，并且可能导致性能降低。

最后，将泛型功能加入进来就能够编写类型安全且高效的基于集合和模式的代码了。泛型提供相当强大的编程能力，但是要求你正确使用它。如果你考虑把 `ArrayList`、`Queue`、`Stack` 和 `Hashtable` 对象转换成其对应的泛型对象，可以阅读一下 1.9 节和 1.10 节中的范例。你将看到这种转换并非总是很简单，有一些原因可能导致你根本不想执行这种转换。

1.1　创建联合类型的结构

1.1.1　问题

你需要创建一种数据类型，其行为方式类似于 C++ 中的联合类型。联合类型主要用于互操作场景，其中非托管代码接受和 / 或返回一个联合类型；我们建议你不要在其他情况下使用它。

1.1.2　解决方案

使用一个结构，并用 StructLayout 特性标记它（在构造函数中指定 LayoutKind.Explicit 布局类型）。此外，利用 FieldOffset 特性标记结构中的每个字段。下面的结构定义了一个联合类型，其中可以存储一个带符号数值。

```
using System.Runtime.IteropServices;
[StructLayoutAttribute(LayoutKind.Explicit)]
struct SignedNumber
{
   [FieldOffsetAttribute(0)]
   public sbyte Num1;
   [FieldOffsetAttribute(0)]
   public short Num2;
   [FieldOffsetAttribute(0)]
   public int Num3;
   [FieldOffsetAttribute(0)]
   public long Num4;
   [FieldOffsetAttribute(0)]
   public float Num5;
   [FieldOffsetAttribute(0)]
   public double Num6;
}
```

下一个结构类似于 SignedNumber 结构，不同之处是除了带符号的数值之外，它还可以包含 String 类型。

```
[StructLayoutAttribute(LayoutKind.Explicit)]
struct SignedNumberWithText
{
   [FieldOffsetAttribute(0)]
   public sbyte Num1;
   [FieldOffsetAttribute(0)]
   public short Num2;
   [FieldOffsetAttribute(0)]
   public int Num3;
   [FieldOffsetAttribute(0)]
   public long Num4;
   [FieldOffsetAttribute(0)]
   public float Num5;
   [FieldOffsetAttribute(0)]
   public double Num6;
   [FieldOffsetAttribute(16)]
```

```
        public string Text1;
    }
```

1.1.3 讨论

联合类型是一种在 C++ 代码中较为常见的结构类型；不过，有一种方式可以使用 C# 中的结构数据类型来复制其结构。联合（union）是一种结构，在内存中的特定位置为该结构接受多种类型。例如，SignedNumber 结构是使用 C# 结构创建的一个联合类型的结构。这种结构可以接受任何类型的带符号的数值类型（sbyte、int 和 long 等），但它只在结构中的同一个位置（同一偏移量）接受这种数字类型。

> 由于 StructLayoutAttribute 可以同时应用于结构和类，在创建联合数据类型时也可以使用类。

注意 FieldOffsetAttribute 将值 0 传递给它的构造函数。这表明这个字段距离结构开始处的偏移量为 0 字节。可以将这个特性与 StructLayoutAttribute 结合使用，手动强制指定结构中的字段开始于什么位置（即每个字段在内存中相对于这个结构开始处的偏移量）。FieldOffsetAttribute 只能与设置为 LayoutKind.Explicit 的 StructLayoutAttribute 一起使用。此外，它不能用于结构内的静态成员。

联合类型可能会带来一些问题，因为几种类型实质上是相互叠加在一起的。最大的问题是如何从联合类型结构中提取正确的数据类型。思考一下，如果你选择在 SignedNumber 结构中存储 long 数值类型的值 long.MaxValue，会发生什么情况。随后，你可能会偶然尝试从这个结构中提取一个 byte 数据类型值。这样操作，你将会只取回这个 long 值中的第一字节。

另一个问题是在正确的偏移位置开始字段。SignedNumberWithText 联合类型在偏移量为 0 的位置叠加了大量带符号的数值数据类型。这个结构中的最后一个字段位于内存中距离这个结构开始处偏移量为 16 字节的位置。如果你意外地把字符串字段 Text1 覆盖在任何其他带符号的数值数据类型之上，在运行时将得到一个异常。基本规则是：允许你把一种值类型叠加在另一种值类型之上，但是不能把一种引用类型叠加于一种值类型之上。如果用以下特性标记 Text1 字段：

```
[FieldOffsetAttribute(14)]
```

就会在运行时引发下面这个异常（注意，编译器不会捕获这个问题）。

```
System.TypeLoadException: Could not load type 'SignedNumberWithText' from
assembly 'CSharpRecipes, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=fe85c3941fbcc4c5' because it contains an object field at
offset 14 that is incorrectly aligned or overlapped by a non-object field.
```

在 C# 中使用复杂的联合类型时，必须保证正确的偏移量。

1.1.4 参考

MSDN 文档中的“StructLayoutAttribute 类”主题。

1.2 使类型可排序

1.2.1 问题

你有一种数据类型，它将存储为 List<T> 或 SortedList<K,V> 的元素。你想使用 List<T>.Sort 方法或者 SortedList<K,V> 的内部排序机制来自定义此数据类型在数组中的排序方式。此外，你可能需要在 SortedList 集合中使用这种类型。

1.2.2 解决方案

例 1-1 演示了如何实现 IComparable<T> 接口。例 1-1 中展示的 Square 类实现了这个接口，使得 List<T> 和 SortedList<K,V> 集合能够排序和查找这些 Square 对象。

例 1-1：通过实现 IComparable<T> 使类型可排序

```
public class Square : IComparable<Square>
{
    public Square(){}

    public Square(int height, int width)
    {
        this.Height = height;
        this.Width = width;
    }

    public int Height { get; set; }

    public int Width { get; set; }

    public int CompareTo(object obj)
    {
        Square square = obj as Square;
        if (square != null)
            return CompareTo(square);
        throw
          new ArgumentException(
              "Both objects being compared must be of type Square.");
    }

    public override string ToString()=>
                       ($"Height: {this.Height}     Width: {this.Width}");

    public override bool Equals(object obj)
    {
        if (obj == null)
            return false;
```

```
            Square square = obj as Square;
            if(square != null)
               return this.Height == square.Height;
            return false;
        }

        public override int GetHashCode()
        {
            return this.Height.GetHashCode() | this.Width.GetHashCode();
        }

        public static bool operator ==(Square x, Square y) => x.Equals(y);
        public static bool operator !=(Square x, Square y) => !(x == y);
        public static bool operator <(Square x, Square y) => (x.CompareTo(y) < 0);
        public static bool operator >(Square x, Square y) => (x.CompareTo(y) > 0);

        public int CompareTo(Square other)
        {
            long area1 = this.Height * this.Width;
            long area2 = other.Height * other.Width;

            if (area1 == area2)
                return 0;
            else if (area1 > area2)
                return 1;
            else if (area1 < area2)
                return -1;
            else
                return -1;
        }
    }
```

1.2.3　讨论

通过在类（或结构）上实现 IComparable<T> 接口，就可以利用 List<T> 和 SortedList<K,V> 类的排序例程。排序算法内置在这些类中；你只需要通过在 IComparable<T>.CompareTo 方法中实现的代码告诉它们如何对你的类进行排序即可。

当调用 List<Square>.Sort 方法对 Square 对象的列表进行排序时，列表是通过 Square 对象的 IComparable<Square> 接口进行排序的。当把对象添加到 SortedList<K,V> 中时，SortedList<K,V> 类的 Add 方法使用这个接口对它们进行排序。

IComparer<T> 设计用于解决如下问题：允许基于不同环境中的不同标准对对象进行排序。这个接口还允许你对其他人编写的类型进行排序。如果你还想按高度对 Square 对象进行排序，就可以创建一个名为 CompareHeight 的新类，如例 1-2 中所示。它也实现了 IComparer<Square> 接口。

例 1-2：通过实现 IComparer 使类型可排序

```
    public class CompareHeight : IComparer<Square>
    {
        public int Compare(object firstSquare, object secondSquare)
```

```
        {
            Square square1 = firstSquare as Square;
            Square square2 = secondSquare as Square;
            if (square1 == null || square2 == null)
                throw (new ArgumentException("Both parameters must be of type Square."));
            else
               return Compare(firstSquare,secondSquare);
        }

        #region IComparer<Square> Members

        public int Compare(Square x, Square y)
        {
            if (x.Height == y.Height)
                return 0;
            else if (x.Height > y.Height)
                return 1;
            else if (x.Height < y.Height)
                return -1;
            else
                return -1;
        }

        #endregion
    }
```

然后将这个类传入 Sort 方法的 IComparer 参数。现在你可以指定以不同的方式对 Square 对象进行排序。比较器中实现的比较方法必须保持一致并应用全局排序，从而使得比较函数声明两个数据项相等时绝对正确，而不是以下情况的结果：一个数据项不大于另一个数据项或者一个数据项不小于另一个数据项。

> 为了获得最佳性能，需要保持 CompareTo 方法短小、高效，因为它将被 Sort 方法调用多次。例如，在对含有 4 个数据项的数组排序时，Compare 方法将被调用 10 次。

例 1-3 中展示的 TestSort 方法演示了如何对 List<Square> 和 SortedList<int,Square> 实例使用 Square 和 CompareHeight 类。

例 1-3：TestSort 方法

```
    public static void TestSort()
    {
        List<Square> listOfSquares = new List<Square>(){
                                    new Square(1,3),
                                    new Square(4,3),
                                    new Square(2,1),
                                    new Square(6,1)};

        // 测试List<String>
        Console.WriteLine("List<String>");
        Console.WriteLine("Original list");
        foreach (Square square in listOfSquares)
```

```
        {
            Console.WriteLine(square.ToString());
        }

        Console.WriteLine();
        IComparer<Square> heightCompare = new CompareHeight();
        listOfSquares.Sort(heightCompare);
        Console.WriteLine("Sorted list using IComparer<Square>=heightCompare");
        foreach (Square square in listOfSquares)
        {
            Console.WriteLine(square.ToString());
        }

        Console.WriteLine();
        Console.WriteLine("Sorted list using IComparable<Square>");
        listOfSquares.Sort();
        foreach (Square square in listOfSquares)
        {
            Console.WriteLine(square.ToString());
        }


        // 测试SortedList
        var sortedListOfSquares = new SortedList<int,Square>(){
                                { 0, new Square(1,3)},
                                { 2, new Square(3,3)},
                                { 1, new Square(2,1)},
                                { 3, new Square(6,1)}};
        Console.WriteLine();
        Console.WriteLine();
        Console.WriteLine("SortedList<Square>");
        foreach (KeyValuePair<int,Square> kvp in sortedListOfSquares)
        {
            Console.WriteLine ($"{kvp.Key} : {kvp.Value}");
        }
    }
```

这些代码的输出如下所示。

```
List<String>
Original list
Height:1 Width:3
Height:4 Width:3
Height:2 Width:1
Height:6 Width:1

Sorted list using IComparer<Square>=heightCompare
Height:1 Width:3
Height:2 Width:1
Height:4 Width:3
Height:6 Width:1

Sorted list using IComparable<Square>
Height:2 Width:1
Height:1 Width:3
```

```
Height:6 Width:1
Height:4 Width:3

SortedList<Square>
0 : Height:1 Width:3
1 : Height:2 Width:1
2 : Height:3 Width:3
3 : Height:6 Width:1
```

1.2.4　参考

范例 1.3（即 1.3 节）；MSDN 文档中的“IComparable<T> 接口”主题。

1.3　使类型可查找

1.3.1　问题

你有一种数据类型，它将存储为 List<T> 中的元素。你想使用 BinarySearch 方法，自定义你的数据类型在列表中的查找方式。

1.3.2　解决方案

使用 IComparable<T> 和 IComparer<T> 接口。范例 1.2（即 1.2 节）中的 Square 类实现了 IComparable<T> 接口，使得 List<T> 和 SortedList<K,V> 集合可以排序和查找 Square 对象的数组和集合。

1.3.3　讨论

通过在类（或结构）上实现 IComparable<T> 接口，就可以利用 List<T> 和 SortedList<K,V> 类的排序例程。排序算法内置在这些类中；你只需要通过在 IComparable<T>.CompareTo 方法中实现的代码告诉它们如何对你的类进行排序即可。

要实现 CompareTo 方法，请参考范例 1.2（即 1.2 节）。

List<T> 类提供了一个 BinarySearch 方法来查找该列表中的元素。列表中的元素会与传递给对象参数中的 BinarySearch 方法的某个对象进行比较。SortedList 类没有 BinarySearch 方法；作为替代，它拥有 ContainsKey 方法，用于对列表中包含的键值执行二分查找。SortedList 类的 ContainsValue 方法在查找值时执行线性查找。这种线性查找使用 SortedList 集合中的元素的 Equals 方法来执行其工作。Compare 和 CompareTo 方法对于 SortedList 类中执行的线性查找不起任何作用，但是它们确实会影响二分查找。

> 为了使用 List<T> 类的 BinarySearch 方法执行准确的查找，首先必须使用 List<T> 的 Sort 方法对其进行排序。此外，如果把一个 IComparer<T> 接口传入给 BinarySearch 方法，还必须把相同的接口传递给 Sort 方法。否则，BinarySearch 方法也许无法找到你正在寻找的对象。

例 1-4 中的 TestSort 方法演示了如何对 List<Square> 和 SortedList<int,Square> 集合实例使用 Square 和 CompareHeight 类。

例 1-4：使类型可查找

```
public static void TestSearch()
{
    List<Square> listOfSquares = new List<Square> {new Square(1,3),
                                                   new Square(4,3),
                                                   new Square(2,1),
                                                   new Square(6,1)};

    IComparer<Square> heightCompare = new CompareHeight();

    // 测试List<Square>
    Console.WriteLine("List<Square>");
    Console.WriteLine("Original list");
    foreach (Square square in listOfSquares)
    {
        Console.WriteLine(square.ToString());
    }

    Console.WriteLine();
    Console.WriteLine("Sorted list using IComparer<Square>=heightCompare");
    listOfSquares.Sort(heightCompare);
    foreach (Square square in listOfSquares)
    {
        Console.WriteLine(square.ToString());
    }

    Console.WriteLine();
    Console.WriteLine("Search using IComparer<Square>=heightCompare");
    int found = listOfSquares.BinarySearch(new Square(1,3), heightCompare);
    Console.WriteLine($"Found (1,3): {found}");

    Console.WriteLine();
    Console.WriteLine("Sorted list using IComparable<Square>");
    listOfSquares.Sort();
    foreach (Square square in listOfSquares)
    {
        Console.WriteLine(square.ToString());
    }

    Console.WriteLine();
    Console.WriteLine("Search using IComparable<Square>");
    found = listOfSquares.BinarySearch(new Square(6,1)); // 使用 IComparable
    Console.WriteLine($"Found (6,1): {found}");

    // 测试SortedList<Square>
    var sortedListOfSquares = new SortedList<int,Square>(){
                                {0, new Square(1,3)},
                                {2, new Square(4,3)},
                                {1, new Square(2,1)},
                                {4, new Square(6,1)}};
  Console.WriteLine();
```

```
    Console.WriteLine("SortedList<Square>");
    foreach (KeyValuePair<int,Square> kvp in sortedListOfSquares)
    {
        Console.WriteLine ($"{kvp.Key} : {kvp.Value}");
    }

    Console.WriteLine();
    bool foundItem = sortedListOfSquares.ContainsKey(2);
    Console.WriteLine($"sortedListOfSquares.ContainsKey(2): {foundItem}");

    // 不要使用IComparer和IComparable
    // -- 使用未重写过的Equals方法实现线性查找
    Console.WriteLine();
    Square value = new Square(6,1);
    foundItem = sortedListOfSquares.ContainsValue(value);
    Console.WriteLine("sortedListOfSquares.ContainsValue " +
                      $"(new Square(6,1)): {foundItem}");
  }
```

这段代码显示的结果如下所示。

```
List<Square>
Original list
Height:1 Width:3
Height:4 Width:3
Height:2 Width:1
Height:6 Width:1

Sorted list using IComparer<Square>=heightCompare
Height:1 Width:3
Height:2 Width:1
Height:4 Width:3
Height:6 Width:1

Search using IComparer<Square>=heightCompare
Found (1,3): 0

Sorted list using IComparable<Square>
Height:2 Width:1
Height:1 Width:3
Height:6 Width:1
Height:4 Width:3

Search using IComparable<Square>
Found (6,1): 2

SortedList<Square>
0 : Height:1 Width:3
1 : Height:2 Width:1
2 : Height:4 Width:3
4 : Height:6 Width:1

sortedListOfSquares.ContainsKey(2): True
sortedListOfSquares.ContainsValue(new Square(6,1)): True
```

1.3.4 参考

范例 1.2（即 1.2 节）；MSDN 文档中的“IComparable<T> 接口”和“IComparer<T> 接口”主题。

1.4 从一个方法返回多个数据项

1.4.1 问题

在许多情况下，从一个方法返回一个值是不够的。你需要一种方式来从一个方法返回不止一个数据项。

1.4.2 解决方案

对充当返回参数的参数使用关键字 out。下面的方法接受一个 inputShape 参数，并通过该值计算 height、width 和 depth。

```
public void ReturnDimensions(int inputShape,
                             out int height,
                             out int width,
                             out int depth)
{
    height = 0;
    width = 0;
    depth = 0;

    // 通过inputShape值计算height、width和depth
}
```

这个方法以如下方式进行调用：

```
// 声明输出参数
int height;
int width;
int depth;

// 调用方法并返回height、width和depth
Obj.ReturnDimensions(1, out height, out width, out depth);
```

另一个方法将返回一个包含所有返回值的类或结构。修改前一个方法，使其返回一个结构，而不是使用 out 参数：

```
public Dimensions ReturnDimensions(int inputShape)
{
    // 默认构造函数自动将结构的成员初始化为0
    Dimensions objDim = new Dimensions();

    // 通过inputShape的值计算objDim.Height、objDim.Width、objDim.Depth……

    return objDim;
}
```

其中 Dimensions 的定义如下所示。

```
public struct Dimensions
{
    public int Height;
    public int Width;
    public int Depth;
}
```

现在以如下方式调用这个方法。

```
// 调用方法并且返回height、width和depth
Dimensions objDim = obj.ReturnDimensions(1);
```

除了从此方法返回一个用户定义的类或结构，也可以用一个 Tuple 对象包含所有的返回值。修改前一个方法，使其返回一个 Tuple。

```
public Tuple<int, int, int> ReturnDimensionsAsTuple(int inputShape)
{
    // 通过inputShape值计算objDim.Height、objDim.Width、objDim.Depth
    // 例如{5, 10, 15}

    // 创建一个包含计算出的值的Tuple
    var objDim = Tuple.Create<int, int, int>(5, 10, 15);

    return (objDim);
}
```

现在以如下方式调用这个方法。

```
// 调用方法并且返回height、width和depth
Tuple<int, int, int> objDim = obj.ReturnDimensions(1);
```

1.4.3　讨论

在方法签名中使用 out 关键字创建一个参数，指示这个参数将由该方法初始化并返回。当需要方法返回多个值时，这个技巧就很有用。一个方法最多只能有一个返回值，但是通过使用 out 关键字，可以把多个参数标记为一个返回值。

要设置一个 out 参数，需要用 out 关键字标记方法签名中的参数，如下所示。

```
public void ReturnDimensions(int inputShape,
                             out int height,
                             out int width,
                             out int depth)
{
    ...
}
```

要调用这个方法，还必须用 out 关键字标记调用方法的参数，如下所示。

```
obj.ReturnDimensions(1, out height, out width, out depth);
```

这个方法中的 out 参数不必初始化；只需声明它们并传入 ReturnDimensions 方法中即可。

不管在调用方法之前是否初始化过它们，在 ReturnDimensions 方法内使用它们之前都必须初始化。即使不通过 ReturnDimensions 方法内的每条路径使用它们，仍然必须初始化它们。这就是这个方法以如下三行代码开始的原因。

```
height = 0;
width = 0;
depth = 0;
```

你可能想知道为什么不能使用 ref 参数代替 out 参数，鉴于它们都允许一个方法改变像这样标记的参数的值。答案是，out 参数使代码有些自文档化。当遇到一个 out 参数时，你知道这个参数充当一个返回值。此外，在把 out 参数传入方法中之前，不需要做额外的工作来初始化它；而 ref 参数则需要这样做。

> 在调用方法时不需要对 out 参数进行封送；相反，在方法把数据返回给调用者时对其封送一次。任何其他调用类型（按值调用或者使用 ref 关键字按引用调用）都要求在两个方向上对值进行封送。在封送场合下使用 out 关键字可以改进远程调用性能。

在仅有少量值需要返回时，out 参数是非常有用的；但是当你遇到需要返回 4 个、5 个、6 个甚至更多的值时，它就变得笨重了。另外一个返回多个值的选项是创建并返回用户定义的类或结构，或者使用 Tuple 打包需要由某个方法返回的所有值。

使用类或结构返回多个值的第一个选项非常直接。只需要像下面这样创建类型（在本例中是该类型是一个结构）即可。

```
public struct Dimensions
{
    public int Height;
    public int Width;
    public int Depth;
}
```

如 1.4.2 节所展示的，将需要的数据填充到这个数据结构的每个字段中，并且从方法中返回它。

与使用用户定义的对象相比，使用 Tuple 的第二个选项更加简洁。可以创建一个 Tuple，用于包含不同类型的任意数量的值。此外，Tuple 中保存的数据是不可变的；一旦通过构造函数或者静态的 Create 方法将数据添加到 Tuple 中，就无法再修改这些数据了。

Tuple 可以接受并包含 8 个独立的值。如里你需要 8 个以上的值，那么需要使用这个特别的 Tuple 类。

```
Tuple<T1, T2, T3, T4, T5, T6, T7, TRest> Class
```

当创建一个包含超过 8 个值的 Tuple 时，你无法使用静态的 Create 方法，而是必须使用 Tuple 类的构造函数。下面的代码展示了如何创建一个包含 10 个整数值的 Tuple。

```
var values = new Tuple<int, int, int, int, int, int, int, Tuple<int, int, int>> (
                     1, 2, 3, 4, 5, 6, 7, new Tuple<int, int, int> (8, 9, 10));
```

当然，你可以继续将更多的 Tuple 添加到每个内嵌的 Tuple 类的最后，以创建你需要的任何大小的 Tuple。

1.4.4 参考

MSDN 文档中的“Tuple 类”和“Tuple<T1, T2, T3, T4, T5, T6, T7, TRest> 类”主题。

1.5 解析命令行参数

1.5.1 问题

你需要应用程序以标准格式（在 1.5.3 节中介绍）接受一个或多个命令行参数。你需要访问和解析传递给应用程序的完整命令行。

1.5.2 解决方案

在例 1-5 中，结合使用以下类来帮你解析命令行参数：Argument、ArgumentDefinition 和 ArgumentSemanticAnalyzer。

例 1-5：Argument 类

```
using System;
using System.Diagnostics;
using System.Linq;
using System.Collections.ObjectModel;

public sealed class Argument
{
    public string Original { get; }
    public string Switch { get; private set; }
    public ReadOnlyCollection<string> SubArguments { get; }
    private List<string> subArguments;
    public Argument(string original)
    {
        Original = original;
        Switch = string.Empty;
        subArguments = new List<string>();
        SubArguments = new ReadOnlyCollection<string>(subArguments);
        Parse();
    }

    private void Parse()
    {
        if (string.IsNullOrEmpty(Original))
        {
            return;
        }
        char[] switchChars = { '/', '-' };
        if (!switchChars.Contains(Original[0]))

        {
```

```
            return;
        }

        string switchString = Original.Substring(1);
        string subArgsString = string.Empty;
        int colon = switchString.IndexOf(':');
        if (colon >= 0)
        {
            subArgsString = switchString.Substring(colon + 1);
            switchString = switchString.Substring(0, colon);
        }
        Switch = switchString;
        if (!string.IsNullOrEmpty(subArgsString))
            subArguments.AddRange(subArgsString.Split(';'));
    }

    // 一组谓词,提供关于参数的有用信息
    // 使用lambda表达式实现
    public bool IsSimple => SubArguments.Count == 0;
    public bool IsSimpleSwitch =>
                !string.IsNullOrEmpty(Switch) && SubArguments.Count == 0;
    public bool IsCompoundSwitch =>
                !string.IsNullOrEmpty(Switch) && SubArguments.Count == 1;
    public bool IsComplexSwitch =>
                !string.IsNullOrEmpty(Switch) && SubArguments.Count > 0;
}

public sealed class ArgumentDefinition
{
     public string ArgumentSwitch { get; }
     public string Syntax { get; }
     public string Description { get; }
     public Func<Argument, bool> Verifier { get; }

     public ArgumentDefinition(string argumentSwitch,
                              string syntax,
                              string description,
                              Func<Argument, bool> verifier)
    {
        ArgumentSwitch = argumentSwitch.ToUpper();
        Syntax = syntax;
        Description = description;
        Verifier = verifier;
    }

    public bool Verify(Argument arg) => Verifier(arg);
}

public sealed class ArgumentSemanticAnalyzer
{

    private List<ArgumentDefinition> argumentDefinitions =
        new List<ArgumentDefinition>();
    private Dictionary<string, Action<Argument>> argumentActions =
        new Dictionary<string, Action<Argument>>();
```

```
public ReadOnlyCollection<Argument> UnrecognizedArguments { get; private set; }
public ReadOnlyCollection<Argument> MalformedArguments { get; private set; }
public ReadOnlyCollection<Argument> RepeatedArguments { get; private set; }

public ReadOnlyCollection<ArgumentDefinition> ArgumentDefinitions =>
       new ReadOnlyCollection<ArgumentDefinition>(argumentDefinitions);

public IEnumerable<string> DefinedSwitches =>
                       from argumentDefinition in argumentDefinitions
                       select argumentDefinition.ArgumentSwitch;

public void AddArgumentVerifier(ArgumentDefinition verifier) =>
    argumentDefinitions.Add(verifier);

public void RemoveArgumentVerifier(ArgumentDefinition verifier)
{
    var verifiersToRemove = from v in argumentDefinitions
                            where v.ArgumentSwitch == verifier.ArgumentSwitch
                            select v;
    foreach (var v in verifiersToRemove)
        argumentDefinitions.Remove(v);
}

public void AddArgumentAction(string argumentSwitch, Action<Argument> action) =>
    argumentActions.Add(argumentSwitch, action);

public void RemoveArgumentAction(string argumentSwitch)
{
    if (argumentActions.Keys.Contains(argumentSwitch))
        argumentActions.Remove(argumentSwitch);
}

public bool VerifyArguments(IEnumerable<Argument> arguments)
{
   // 没有任何参数进行验证,失败
   if (!argumentDefinitions.Any())

        return false;

    // 确认是否存在任一未定义的参数
    this.UnrecognizedArguments =
             ( from argument in arguments
               where !DefinedSwitches.Contains(argument.Switch.ToUpper())
               select argument).ToList().AsReadOnly();

    if (this.UnrecognizedArguments.Any())
        return false;

    //检查开关与某个已知开关匹配但是检查格式是否正确
    //的谓词为false的所有参数
    this.MalformedArguments = ( from argument in arguments
                                join argumentDefinition in argumentDefinitions
                                on argument.Switch.ToUpper() equals
                                    argumentDefinition.ArgumentSwitch
```

```
                                    where !argumentDefinition.Verify(argument)
                                    select argument).ToList().AsReadOnly();

    if (this.MalformedArguments.Any())
        return false;

    //将所有参数按照开关进行分组,统计每个组的数量，
    //并选出包含超过一个元素的所有组，
    //然后我们获得一个包含这些数据项的只读列表
    this.RepeatedArguments =
            (from argumentGroup in
                from argument in arguments
                where !argument.IsSimple
                group argument by argument.Switch.ToUpper()
            where argumentGroup.Count() > 1
            select argumentGroup).SelectMany(ag => ag).ToList().AsReadOnly();

    if (this.RepeatedArguments.Any())
    return false;

    return true;
}

public void EvaluateArguments(IEnumerable<Argument> arguments)
{
    //此时只需应用每个动作：
    foreach (Argument argument in arguments)
        argumentActions[argument.Switch.ToUpper()](argument);
}

public string InvalidArgumentsDisplay()
{
    StringBuilder builder = new StringBuilder();
    builder.AppendFormat($"Invalid arguments: {Environment.NewLine}");
    // 添加未识别的参数

    FormatInvalidArguments(builder, this.UnrecognizedArguments,
        "Unrecognized argument: {0}{1}");

    // 添加格式不正式的参数
    FormatInvalidArguments(builder, this.MalformedArguments,
        "Malformed argument: {0}{1}");

   // 对于重复的参数,我们想要将其分组以用于显示，
   // 因此通过开关分组并且将其添加到正在构建的字符串
   var argumentGroups = from argument in this.RepeatedArguments
                        group argument by argument.Switch.ToUpper() into ag
                        select new { Switch = ag.Key, Instances = ag};

   foreach (var argumentGroup in argumentGroups)
   {
       builder.AppendFormat($"Repeated argument:
                              {argumentGroup.Switch}{Environment.NewLine}");
       FormatInvalidArguments(builder, argumentGroup.Instances.ToList(),
           "\t{0}{1}");
```

```
        }
        return builder.ToString();
    }

    private void FormatInvalidArguments(StringBuilder builder,
       IEnumerable<Argument> invalidArguments, string errorFormat)
    {
         if (invalidArguments != null)
         {
             foreach (Argument argument in invalidArguments)
             {
                 builder.AppendFormat(errorFormat,
                     argument.Original, Environment.NewLine);
             }
         }
     }
}
```

如何使用这些类为应用程序处理命令行？方法如下所示。

```
public static void Main(string[] argumentStrings)
{
    var arguments = (from argument in argumentStrings
        select new Argument(argument)).ToArray();

    Console.Write("Command line: ");
    foreach (Argument a in arguments)
    {
        Console.Write($"{a.Original} ");
    }
    Console.WriteLine("");

    ArgumentSemanticAnalyzer analyzer = new ArgumentSemanticAnalyzer();
    analyzer.AddArgumentVerifier(
        new ArgumentDefinition("output",
            "/output:[path to output]",
            "Specifies the location of the output file.",
            x => x.IsCompoundSwitch));
    analyzer.AddArgumentVerifier(
        new ArgumentDefinition("trialMode",
            "/trialmode",
            "If this is specified it places the product into trial mode",
            x => x.IsSimpleSwitch));
    analyzer.AddArgumentVerifier(
        new ArgumentDefinition("DEBUGOUTPUT",
            "/debugoutput:[value1];[value2];[value3]",
            "A listing of the files the debug output " +
            "information will be written to",
            x => x.IsComplexSwitch));
    analyzer.AddArgumentVerifier(
        new ArgumentDefinition("",
            "[literal value]",
            "A literal value",
            x => x.IsSimple));
```

```
    if (!analyzer.VerifyArguments(arguments))
    {
        string invalidArguments = analyzer.InvalidArgumentsDisplay();
        Console.WriteLine(invalidArguments);
        ShowUsage(analyzer);
        return;
    }

    // 设置命令行解析结果的容器
    string output = string.Empty;
    bool trialmode = false;
    IEnumerable<string> debugOutput = null;
    List<string> literals = new List<string>();

    //我们想对每一个解析出的参数应用一个动作，
    //因此将它们添加到分析器
    analyzer.AddArgumentAction("OUTPUT", x => { output = x.SubArguments[0]; });
    analyzer.AddArgumentAction("TRIALMODE", x => { trialmode = true; });
    analyzer.AddArgumentAction("DEBUGOUTPUT", x =>
                                          { debugOutput = x.SubArguments;
    });

    analyzer.AddArgumentAction("", x=>{literals.Add(x.Original);});

    // 检查参数并运行动作
    analyzer.EvaluateArguments(arguments);

    // 显示结果
    Console.WriteLine("");
    Console.WriteLine($"OUTPUT: {output}");
    Console.WriteLine($"TRIALMODE: {trialmode}");
    if (debugOutput != null)
    {
        foreach (string item in debugOutput)

        {
             Console.WriteLine($"DEBUGOUTPUT: {item}");
        }
    }
    foreach (string literal in literals)
    {
        Console.WriteLine($"LITERAL: {literal}");
    }
}

public static void ShowUsage(ArgumentSemanticAnalyzer analyzer)
{
    Console.WriteLine("Program.exe allows the following arguments:");
    foreach (ArgumentDefinition definition in analyzer.ArgumentDefinitions)
    {
        Console.WriteLine($"\t{definition.ArgumentSwitch}:
                         ({definition.Description}){Environment.NewLine}
                         \tSyntax: {definition.Syntax}");
    }
}
```

1.5.3 讨论

在解析命令行参数之前，必须明确选用一种通用格式。本范例中使用的格式遵循用于 Visual C#.NET 语言编译器的命令行格式。使用的格式定义如下所示。

- 通过一个或多个空白字符分隔命令行参数。
- 每个参数可以以一个 - 或 / 字符开头，但不能同时以这两个字符开头。如果不以其中一个字符开头，就把参数视为一个字面量，比如文件名。
- 以 - 或 / 字符开头的参数可被划分为：以一个选项开关开头，后接一个冒号，再接一个或多个用 ; 字符分隔的参数。命令行参数 `-sw:arg1;arg2;arg3` 可被划分为一个选项开关（`sw`）和三个参数（`arg1`、`arg2` 和 `arg3`）。注意，在完整的参数中不应该有任何空格，否则运行时命令行解析器将把参数分拆为两个或更多的参数。
- 用双引号包裹住的字符串（如 `"c:\test\file.log"`）会去除双引号。这是操作系统解释传入应用程序中的参数时的一项功能。
- 不会去除单引号。
- 要保留双引号，可在双引号字符前放置 \ 转义序列字符。
- 仅当 \ 字符后面接着双引号时，才将 \ 字符作为转义序列字符处理；在这种情况下，只会显示双引号。
- ^ 字符被运行时解析器作为特殊字符处理。

幸运的是，在应用程序接收各个解析出的参数之前，运行时命令行解析器可以处理其中大部分任务。

运行时命令行解析器把一个包含每个解析过的参数的 `string[]` 传递给应用程序的入口点。入口点可以采用以下形式之一。

```
public static void Main()
public static int Main()
public static void Main(string[] args)
public static int Main(string[] args)
```

前两种形式不接受参数，但是后两种形式接受解析过的命令行参数的数组。注意，静态属性 `Environment.CommandLine` 将返回一个字符串，其中包含完整的命令行；静态方法 `Environment.GetCommandLineArgs` 将返回一个字符串数组，其中包含解析过的命令行参数。

1.5.2 节介绍的三个类涉及命令行参数的各个阶段。

- `Argument`

  封装一个命令行参数并负责解析该参数。

- `ArgumentDefinition`

  定义一个对当行命令行有效的参数。

- `ArgumentSemanticAnalyzer`

  基于设置的 `ArgumentDefinition` 进行参数的验证和获取。

把以下命令行参数传入这个应用程序中：

```
MyApp c:\input\infile.txt -output:d:\outfile.txt -trialmode
```

将得到以下解析过的选项开关和参数。

```
Command line: c:\input\infile.txt -output:d:\outfile.txt -trialmode
OUTPUT: d:\outfile.txt
TRIALMODE: True
LITERAL: c:\input\infile.txt
```

如果你没有正确地输入命令行参数，比如忘记了向 `-output` 选项开关添加参数，得到的输出将如下所示。

```
Command line: c:\input\infile.txt -output: -trialmode
Invalid arguments:
Malformed argument: -output

Program.exe allows the following arguments:
       OUTPUT: (Specifies the location of the output file.)
       Syntax: /output:[path to output]
       TRIALMODE: (If this is specified, it places the product into trial mode)
       Syntax: /trialmode
       DEBUGOUTPUT: (A listing of the files the debug output information will be
                     written to)
       Syntax: /debugoutput:[value1];[value2];[value3]
       : (A literal value)
       Syntax: [literal value]
```

在这段代码中有几个值得指出的地方。

每个 `Argument` 实例都需要能确定它自身的某些事项。相应地，作为 `Argument` 的属性暴露了一组谓词，告诉我们这个 `Argument` 的一些有用信息。`ArgumentSemanticAnalyzer` 将使用这些属性来确定参数的特征。

```
public bool IsSimple => SubArguments.Count == 0;
public bool IsSimpleSwitch =>
            !string.IsNullOrEmpty(Switch) && SubArguments.Count == 0;
public bool IsCompoundSwitch =>
            !string.IsNullOrEmpty(Switch) && SubArguments.Count == 1;
public bool IsComplexSwitch =>
            !string.IsNullOrEmpty(Switch) && SubArguments.Count > 0;
```

关于 lambda 表达式的更多信息请参考 4.0 节。范例 1.16（即 1.16 节）中也包含了使用 lambda 表达式来实现闭包的相关讨论。

这段代码有多处在 LINQ 查询的结果上调用了 `ToArray` 或 `ToList` 方法。

```
var arguments = (from argument in argumentStrings
                 select new Argument(argument)).ToArray();
```

这是由于查询结果是延迟执行的。这不仅意味着将以迟缓方式来计算结果，而且意味着每

次访问结果时都要重新计算它们。使用 ToArray 或 ToList 方法会强制积极计算结果，生成一份不需要在每次使用时都重新计算的副本。查询逻辑并不知道正在操作的集合是否发生了变化，因此每次都必须重新计算结果，除非使用这些方法创建出一份“即时”副本。

为了验证这些参数是否正确，必须创建 ArgumentDefinition，并将每个可接受的参数类型与 ArgumentSemanticAnalyzer 相关联，代码如下所示。

```
ArgumentSemanticAnalyzer analyzer = new ArgumentSemanticAnalyzer();
analyzer.AddArgumentVerifier(
        new ArgumentDefinition("output",
            "/output:[path to output]",
            "Specifies the location of the output file.",
            x => x.IsCompoundSwitch));
analyzer.AddArgumentVerifier(
        new ArgumentDefinition("trialMode",
            "/trialmode",
            "If this is specified it places the product into trial mode",
            x => x.IsSimpleSwitch));
analyzer.AddArgumentVerifier(
        new ArgumentDefinition("DEBUGOUTPUT",
            "/debugoutput:[value1];[value2];[value3]",
            "A listing of the files the debug output " +
            "information will be written to",
            x => x.IsComplexSwitch));
analyzer.AddArgumentVerifier(
        new ArgumentDefinition("",
            "[literal value]",
            "A literal value",
            x => x.IsSimple));
```

每个 ArgumentDefinition 都包含 4 个部分：参数选项开关、显示参数语法的字符串、参数说明以及用于验证参数的验证谓词。这些信息可以用于验证参数，如下所示。

```
//检查开关与某个已知开关匹配但是检查格式是否正确
//的谓词为false的所有参数
this.MalformedArguments = ( from argument in arguments
                            join argumentDefinition in argumentDefinitions
                              on argument.Switch.ToUpper() equals
                                  argumentDefinition.ArgumentSwitch
                              where !argumentDefinition.Verify(argument)
                              select argument).ToList().AsReadOnly();
```

ArgumentDefinition 还允许为程序编写一个使用说明方法。

```
public static void ShowUsage(ArgumentSemanticAnalyzer analyzer)
{
    Console.WriteLine("Program.exe allows the following arguments:");
    foreach (ArgumentDefinition definition in analyzer.ArgumentDefinitions)
    {
        Console.WriteLine("\t{0}: ({1}){2}\tSyntax: {3}",
            definition.ArgumentSwitch, definition.Description,
            Environment.NewLine,definition.Syntax);
    }
}
```

为了获取参数的值以便使用它们，需要从解析过的参数中提取信息。对于解决方案示例，我们需要以下信息。

```
// 设置命令行解析结果的容器
string output = string.Empty;
bool trialmode = false;
IEnumerable<string> debugOutput = null;
List<string> literals = new List<string>();
```

如何填充这些值？对于每个参数，都需要一个与之关联的动作，以确定如何从 `Argument` 实例获得值。每个动作就是一个谓词，这使得这种方式非常强大，因为你在这里可以使用任何谓词。下面的代码说明如何定义这些 `Argument` 动作并将其与 `ArgumentSemanticAnalyzer` 相关联。

```
//对于每一个解析出的参数,我们想要对其应用一个动作,
//因此将它们添加到分析器
analyzer.AddArgumentAction("OUTPUT", x => { output = x.SubArguments[0]; });
analyzer.AddArgumentAction("TRIALMODE", x => { trialmode = true; });
analyzer.AddArgumentAction("DEBUGOUTPUT", x =>
                                           { debugOutput = x.SubArguments;});
analyzer.AddArgumentAction("", x=>{literals.Add(x.Original);});
```

现在已经建立了所有的动作，就可以对 `ArgumentSemanticAnalyzer` 应用 `EvaluateArguments` 方法来获取值，代码如下所示。

```
// 检查参数并运行动作
analyzer.EvaluateArguments(arguments);
```

现在通过执行动作填充了值，并且可以利用这些值来运行程序，代码如下所示。

```
// 传入参数值并运行程序
Program program = new Program(output, trialmode, debugOutput, literals);
program.Run();
```

如果在验证参数时使用 LINQ 来查询未识别的、格式错误的或者重复的实参（argument），其中任何一项都会导致形参（parameter）无效。

```
public bool VerifyArguments(IEnumerable<Argument> arguments)
{
    // 没有任何参数进行验证,失败
    if (!argumentDefinitions.Any())
        return false;

    // 确认是否存在任一未定义的参数
    this.UnrecognizedArguments =
                ( from argument in arguments
                  where !DefinedSwitches.Contains(argument.Switch.ToUpper())
                  select argument).ToList().AsReadOnly();

    if (this.UnrecognizedArguments.Any())
        return false;

    //检查开关与某个已知开关匹配但是检查格式是否正确
    //的谓词为false的所有参数
```

```
        this.MalformedArguments = ( from argument in arguments
                                    join argumentDefinition in argumentDefinitions
                                    on argument.Switch.ToUpper() equals
                                        argumentDefinition.ArgumentSwitch
                                    where !argumentDefinition.Verify(argument)
                                    select argument).ToList().AsReadOnly();
        if (this.MalformedArguments.Any())
            return false;

        //将所有参数按照开关进行分组,统计每个组的数量,
        //并选出包含超过一个元素的所有组,
        //然后我们获得一个包含这些数据项的只读列表
        this.RepeatedArguments =
                    (from argumentGroup in
                        from argument in arguments
                        where !argument.IsSimple
                        group argument by argument.Switch.ToUpper()
                    where argumentGroup.Count() > 1
                    select argumentGroup).SelectMany(ag => ag).ToList().AsReadOnly();

        if (this.RepeatedArguments.Any())
            return false;

        return true;
    }
```

与 LINQ 出现之前通过多重嵌套循环、`switch` 语句、`IndexOf` 方法及其他机制实现同样功能的代码相比，上述使用 LINQ 的代码更加易于理解每一个验证阶段。每个查询都用问题领域的语言简洁地指出了它在尝试执行什么任务。

> LINQ 旨在帮助解决那些必须排序、查找、分组、筛选和投影数据的问题。请使用它！

1.5.4 参考

MSDN 文档中的“Main”和“命令行参数”主题。

1.6 在运行时初始化常量字段

1.6.1 问题

标记为 `const` 的字段只能在编译时初始化。你需要在运行时而不是在编译时将一个字段初始化为一个有效值。然后在应用程序剩余的生命期内，这个字段必须像一个常量字段那样工作。

1.6.2 解决方案

在代码中声明一个常量值时有两种选择。你可以使用 readonly 字段或 const 字段，每种方式都有其优缺点。不过，如果你需要在运行时初始化一个常量字段，就必须使用 readonly 字段。

```
public class Foo
{
    public readonly int bar;

    public Foo() {}

    public Foo(int constInitValue)
    {
        bar = constInitValue;
    }

    // 类的其他部分
}
```

使用 const 字段无法做到这一点。const 字段只能在编译时初始化。

```
public class Foo
{
    public const int bar;     // 这一行造成一个编译时错误

    public Foo() {}

    public Foo(int constInitValue)
    {
        bar = constInitValue; //这一行同样造成一个编译时错误
    }
    // 类的其他部分
}
```

1.6.3 讨论

readonly 字段只允许在运行时在构造函数中执行初始化，而 const 字段必须在编译时进行初始化。因此，为了让一个必须为常量的字段在运行时初始化，唯一的方式是实现一个 readonly 字段。

只有两种方式可用于初始化一个 readonly 字段。第一种方式是向字段自身添加一个初始化器，代码如下所示。

```
public readonly int bar = 100;
```

第二种方式是通过一个构造函数初始化 readonly 字段。1.6.2 节中的代码演示了这种方法。如果查看下面的类：

```
public class Foo
{
    public readonly int x;
    public const int y = 1;
```

```
        public Foo() {}
        public Foo(int roInitValue)
        {
            x = roInitValue;
        }

        // 类的其他部分
    }
```

你会看到它被编译成下面的中间语言（intermediate language，IL）：

```
.class auto ansi nested public beforefieldinit Foo
       extends [mscorlib]System.Object       {
.field public static literal int32 y = int32(0x00000001) //<<-- const field
.field public initonly int32 x                          //<<-- readonly field
.method public hidebysig specialname rtspecialname
        instance void  .ctor(int32 roInitValue) cil managed
{
  // Code size       16 (0x10)
  .maxstack  8
  IL_0000:  ldarg.0
  IL_0001:  call      instance void [mscorlib]System.Object::.ctor()
  IL_0006:  nop
  IL_0007:  nop
  IL_0008:  ldarg.0
  IL_0009:  ldarg.1
  IL_000a:  stfld     int32 CSharpRecipes.ClassesAndGenerics/Foo::x
  IL_000f:  ret
} // end of method Foo::.ctor
.method public hidebysig specialname rtspecialname
        instance void  .ctor() cil managed
{
  // Code size       9 (0x9)
  .maxstack  8
  IL_0000:  ldarg.0
  IL_0001:  call      instance void [mscorlib]System.Object::.ctor()
  IL_0006:  nop
  IL_0007:  nop
  IL_0008:  ret
} // end of method Foo::.ctor
} // End of class Foo
```

注意 `const` 字段被编译成一个静态字段，`readonly` 字段被编译成一个实例字段。因此，只需要类名就可以访问一个 `const` 字段。

> 对于使用 `const` 字段的一个常见争论是，它们并不像 `readonly` 字段那样支持版本化。如果重新构建一个定义了 `const` 字段的组件，并且该 `const` 字段的值在之后的版本中发生了改变，那么使用旧版本构建的任何其他组件都不会获得新的值。只要一个字段将来有可能发生改变，就不要把它声明为一个 `const` 字段。

下面的代码显示了如何使用一个 `readonly` 实例字段。

```
Foo obj1 = new Foo(100);
Console.WriteLine(obj1.bar);
```

1.6.4 参考

MSDN 文档中的“const”和“readonly”关键字。

1.7 构建可克隆的类

1.7.1 问题

你需要一种方法对可能引用其他类型的数据类型进行浅克隆操作、深克隆操作或者同时执行这两种操作，但是不应该使用 ICloneable 接口，因为它违反了 .NET Framework 设计准则。

1.7.2 解决方案

为了解决使用 ICloneable 的问题，创建另外两个接口 IShallowCopy<T> 和 IDeepCopy<T> 来建立一种复制模式，代码如下所示。

```
public interface IShallowCopy<T>
{
    T ShallowCopy();
}
public interface IDeepCopy<T>
{
    T DeepCopy();
}
```

浅复制意味着所复制对象的字段将引用与原始对象相同的对象。为了允许进行浅复制，可在类中实现 IShallowCopy<T> 接口，代码如下所示。

```
using System;
using System.Collections;
using System.Collections.Generic;

public class ShallowClone : IShallowCopy<ShallowClone>
{
    public int Data = 1;
    public List<string> ListData = new List<string>();
    public object ObjData = new object();

    public ShallowClone ShallowCopy() => (ShallowClone)this.MemberwiseClone();
}
```

深复制（或称克隆）意味着所复制对象的字段将引用原始对象的字段的新副本。为了进行深复制，可在类中实现 IDeepCopy<T> 接口，代码如下所示。

```
using System;
using System.Collections;
```

```
using System.Collections.Generic;
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;

[Serializable]
public class DeepClone : IDeepCopy<DeepClone>
{
    public int data = 1;
    public List<string> ListData = new List<string>();
    public object objData = new object();

    public DeepClone DeepCopy()
    {
        BinaryFormatter BF = new BinaryFormatter();
        MemoryStream memStream = new MemoryStream();

        BF.Serialize(memStream, this);
        memStream.Flush();
        memStream.Position = 0;

        return (DeepClone)BF.Deserialize(memStream);
    }
}
```

要同时支持浅复制和深复制方法，可同时实现这两个接口，代码如下所示。

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;

[Serializable]
public class MultiClone : IShallowCopy<MultiClone>,
                          IDeepCopy<MultiClone>
{
    public int data = 1;
    public List<string> ListData = new List<string>();
    public object objData = new object();

    public MultiClone ShallowCopy() => (MultiClone)this.MemberwiseClone();

    public MultiClone DeepCopy()
    {
        BinaryFormatter BF = new BinaryFormatter();
        MemoryStream memStream = new MemoryStream();

        BF.Serialize(memStream, this);
        memStream.Flush();
        memStream.Position = 0;

        return (MultiClone)BF.Deserialize(memStream);
    }
}
```

1.7.3　讨论

.NET Framework 中包含一个名为 `ICloneable` 的接口，它最初被设计为在 .NET 中实现克隆的方法。设计建议现在不再在任何公开 API 中使用这个接口，因为它容易将自身导向不同的解释。此接口看起来如下所示。

```
public interface ICloneable
{
    object Clone();
}
```

注意此接口只有一个方法 `Clone`，它返回一个对象。该克隆是对象的浅副本还是深副本呢？无法通过该接口得知这一点，因为实现可以选择任何一个方式。这就是不应该再使用它，而是引入 `IShallowCopy<T>` 和 `IDeepCopy<T>` 接口的原因。

克隆操作能够创建出类型实例的一个准确副本（克隆）。克隆可能采用两种形式之一：浅复制和深复制。浅复制相对容易一些，它对涉及复制的对象调用 `ShallowCopy` 方法。

在原始对象中，引用类型的字段像值类型的字段那样进行复制。例如，如果原始对象包含一个 `StreamWriter` 类型的字段，克隆的对象将指向原始对象的 `StreamWriter` 的同一个实例，并没有创建新对象。

> 在执行克隆操作时无需处理静态字段。每个应用程序域中的每个类的每个静态字段只会保留一个内存位置。克隆出的对象与原始对象访问相同的静态字段。

对浅复制的支持是通过 `Object` 类的 `MemberwiseClone` 方法来实现的，`Object` 类充当了所有 .NET 类的基类。因此，下面的代码通过 `Clone` 方法允许创建和返回一个浅复制。

```
public ShallowClone ShallowCopy() => (ShallowClone)this.MemberwiseClone();
```

克隆一个对象的另一种方式是创建深复制。就像浅复制那样，深复制将创建原始对象的一个副本。不同的是，深复制还会创建原始对象中每个引用类型的字段的单独副本。因此，如果原始对象包含一个 `StreamWriter` 类型的字段，复制的对象也会包含一个 `StreamWriter` 类型的字段，但是复制对象的 `StreamWriter` 字段将指向一个新的 `StreamWriter` 对象，而不是原始对象的 `StreamWriter` 对象。

.NET Framework 没有直接提供对深复制的支持，但是下面的代码提供了一种实现深复制的简单方式。

```
BinaryFormatter BF = new BinaryFormatter();
MemoryStream memStream = new MemoryStream();

BF.Serialize(memStream, this);
memStream.Flush();
memStream.Position = 0;

return (BF.Deserialize(memStream));
```

总而言之，这使用二进制序列化将原始对象序列化到一个内存流中，然后将其反序列化到一个新对象中，并将该对象返回给调用者。在调用 Deserialize 方法之前将内存流指针重新定位到流的开始处是十分重要的；否则就会引发一个异常，指示序列化的对象中不包含任何数据。

使用对象序列化执行深复制时，不必修改执行深复制的代码就能改下层的对象。如果你手动执行深复制，就必须对原始对象的每个实例字段创建新实例，并把新实例复制到克隆的对象。这是一件非常琐碎的事情。如果修改了原始对象的字段，你必须修改深复制的代码以反映出这些修改。使用序列化可以依靠序列化器动态查找和序列化对象中包含的所有字段。如果修改了对象，序列化器仍然不需要修改就可以进行深复制。

你可能想手动执行深复制的一个原因是，仅当对象中的一切都可序列化时，本范例中介绍的序列化技术才会正确工作。当然，手动复制有时也于事无补，因为有些对象天生就是不可复制的。假设你有一个网络管理应用，其中一个对象代表网络上的一台特定打印机。当你复制它时会指望它做什么呢？传真一份订购单以购买一台新的打印机吗？

深复制与生俱来的一个问题是在具有循环引用的嵌套数据结构上执行深复制。本范例使得处理循环引用成为可能，尽管这仍是一个难题。因此，事实上，如果你使用本范例中的方法，就无需避免循环引用。

1.7.4　参考

Krzysztof Cwalina 和 Brad Abrams 所著的《.NET 设计规范：约定、惯用法与模式（第 2 版）》；MSDN 文档中的“Object.MemberwiseClone 方法”主题。

1.8　确保对象的处置

1.8.1　问题

当一个对象的工作完成或者超出作用域时，你需要采用一种方式确保一些处理得到执行。

1.8.2　解决方案

使用 using 语句，代码如下所示。

```
using System;
using System.IO;

// ...

using(FileStream FS = new FileStream("Test.txt", FileMode.Create))
{
    FS.WriteByte((byte)1);
    FS.WriteByte((byte)2);
    FS.WriteByte((byte)3);

    using(StreamWriter SW = new StreamWriter(FS))
```

```
        {
            SW.WriteLine("some text.");
        }
    }
```

1.8.3 讨论

using 语句非常易于使用，并且可以避免编写多余代码的麻烦。如果解决方案中没有使用 using 语句，将会如下所示。

```
FileStream FS = new FileStream("Test.txt", FileMode.Create);
try
{
    FS.WriteByte((byte)1);
    FS.WriteByte((byte)2);
    FS.WriteByte((byte)3);

    StreamWriter SW = new StreamWriter(FS);

    try
    {
       SW.WriteLine("some text.");
    }
    finally
    {
        if (SW != null)
        {
            ((IDisposable)SW).Dispose();
        }
    }
}
finally
{
   if (FS != null)
   {
       ((IDisposable)FS).Dispose();
   }
}
```

关于 using 语句，需要注意以下几点。

- 存在一个 using 指令，如下所示。应将其与 using 语句区分开。这可能会使初次接触这一语言的开发人员混淆。

```
using System.IO;
```

- using 语句的子句中定义的变量都必须具有相同的类型，并且必须具有一个初始化器。不过，因为可以在单个代码块前使用多个 using 语句，所以这并不是一个重大的限制。
- using 子句中定义的变量在 using 语句体中被认为是只读的。这可以阻止开发人员在尝试处置变量最初引用的对象时无意中将变量转变成引用不同的对象，避免引发问题。
- 不应该在 using 块外声明变量，然后在 using 子句内初始化它。

下面的代码说明了最后一点。

```
FileStream FS;
using(FS = new FileStream("Test.txt", FileMode.Create))
{
    FS.WriteByte((byte)1);
    FS.WriteByte((byte)2);
    FS.WriteByte((byte)3);

    using(StreamWriter SW = new StreamWriter(FS))
    {
        SW.WriteLine("some text.");
    }
}
```

对于这段示例代码来说，不会有任何问题。但是，要考虑到变量 `FS` 是可以在 `using` 块外使用的。实际上，可以将这段代码修改为下面这样。

```
FileStream FS;
using(FS = new FileStream("Test.txt", FileMode.Create))
{
    FS.WriteByte((byte)1);
    FS.WriteByte((byte)2);
    FS.WriteByte((byte)3);

    using(StreamWriter SW = new StreamWriter(FS))
    {
        SW.WriteLine("some text.");
    }
}
FS.WriteByte((byte)4);
```

这段代码可以编译，但会在这个代码段的最后一行上引发一个 `ObjectDisposedException`，因为已经对 `FS` 对象调用了 `Dispose` 方法。对象此时还没有被垃圾回收，仍然以被处置过的状态存在于内存中。

1.8.4　参考

MSDN 文档中的"Cleaning Up Unmanaged Resources""`IDisposable` 接口""Using `foreach` with Collections"和"实现 `Finalize` 和 `Dispose` 以清理非托管资源"主题。

1.9　确定何时何处使用泛型

1.9.1　问题

你想在新项目中使用泛型类型或者把现有项目中的非泛型类型转换成它们对应的泛型类型。不过，你并不真正明白为什么要这样做，也不知道应该把哪些非泛型类型转换成泛型类型。

1.9.2 解决方案

在决定何时何处使用泛型类型时，需要考虑以下几点。

- 你的类型将包含或操作多种不同的未确定数据类型（例如，一种集合类型）吗？如果是，那么与创建非泛型类型相比，创建泛型类型会有几个好处。如果你的类型只操作某种特定的类型，那么可能不需要创建泛型类型。
- 如果你的类型在值类型上操作，那么将会发生装箱和拆箱操作。你应该考虑使用泛型来避免装箱和拆箱操作带来的性能损失。
- 与泛型关联的更强大的类型检查有助于更快地（即在编译时而不是在运行时）找出错误，从而缩短错误修正周期。
- 由于你编写了多个类来处理不同的数据类型（例如，使用一个专门的 `ArrayList` 只存储 `StreamReader`，并使用另一个 `ArrayList` 只存储 `StreamWriter`），这导致你的代码产生“代码膨胀”了吗？更容易的做法是，编写一次代码，使其适用于所操作的所有数据类型。
- 泛型可以使得代码更清晰。通过消除代码膨胀并强制对类型执行更强大的类型检查，可以使代码更易于阅读和理解。

1.9.3 讨论

在大多数情况下，你的代码将受益于使用泛型类型。泛型可以实现更高效的代码重用，更快速的性能，更强大的类型检查和更易于阅读的代码。

1.9.4 参考

MSDN 文档中的“泛型概述”和“泛型的优点”主题。

1.10 理解泛型类型

1.10.1 问题

你需要理解 .NET 类型如何适用于泛型，以及泛型 .NET 类型与常规 .NET 类型有着怎样的区别。

1.10.2 解决方案

可以用两个快速的试验来展示常规 .NET 类型和泛型 .NET 类型之间的区别。在深入代码之前，如果你对泛型不熟悉，可以先跳到具体解释泛型的 1.10.3 节，之后再回到本部分。

当定义一个常规 .NET 类型时，它看起来就像是例 1-6 中定义的 `FixedSizeCollection` 类型。

例 1-6：`FixedSizeCollection`（一种常规 .NET 类型）

```
public class FixedSizeCollection
{
    /// <summary>
    /// 构造函数,增加静态计数器的值
```

```
/// 设置数据项最大数量
/// </summary>
/// <param name="maxItems"></param>
public FixedSizeCollection(int maxItems)
{
     FixedSizeCollection.InstanceCount++;
     this.Items = new object[maxItems];
}
/// <summary>
/// 将一个未知类型的数据项添加到类中
/// object可以包含任何类型
/// </summary>
/// <param name="item">要添加的数据项</param>
/// <returns>新添加的数据项的索引</returns>
public int AddItem(object item)
{
    if (this.ItemCount < this.Items.Length)
    {
        this.Items[this.ItemCount] = item;
        return this.ItemCount++;
    }
    else
        throw new Exception("Item queue is full");
}

/// <summary>
/// 从类中获得一个数据项
/// </summary>
/// <param name="index">要获取的数据项的索引</param>
/// <returns>object类型的一个数据项</returns>
public object GetItem(int index)
{
    if (index >= this.Items.Length &&
        index >= 0)
        throw new ArgumentOutOfRangeException(nameof(index));
    return this.Items[index];
}

#region Properties
/// <summary>
/// 静态的实例计数器，用于标准类型
/// </summary>
public static int InstanceCount { get; set; }

/// <summary>
/// 类中包含的数据项数量
/// </summary>
public int ItemCount { get; private set; }

/// <summary>
/// 类中包含的数据项
/// </summary>
private object[] Items { get; set; }
#endregion // Properties
```

```
    /// <summary>
    /// 重写ToString以提供类的详细信息
    /// </summary>
    /// <returns>包含类详细信息、格式化过的字符串</returns>
    public override string ToString() =>
                $"There are {FixedSizeCollection.InstanceCount.ToString()}
                instances of {this.GetType().ToString()} and this instance
                contains {this.ItemCount} items...";
}
```

FixedSizeCollection 拥有一个静态整型属性变量 InstanceCount，在实例构造函数中递增；还包含一个 ToString() 替代，用于输出这个 AppDomain.FixedSizeCollection 中存在多少个 FixedSizeCollection 的实例。此外，此集合类还包含一个 objects(Items) 的数组，其大小由传入构造函数的项数量确定。FixedSizeCollection 还实现了两个方法（AddItem 和 GetItem），用于添加和获取数据项；实现了一个只读属性（ItemCount），用于获取数组中当前的数据项数量。

FixedSizeCollection<T> 类型是一种泛型 .NET 类型，它具有相同的静态属性字段 InstanceCount、统计实例化次数的实例构造函数，以及重写的 ToString() 方法，用于指出这种类型的实例有多少个。FixedSizeCollection<T> 还具有一个 Items 数组属性，以及与 FixedSizeCollection 类对应的方法，如例 1-7 所示。

例 1-7：FixedSizeCollection<T>（一种泛型 .NET 类型）

```
/// <summary>
/// 演示实例计数的泛型类
/// </summary>
/// <typeparam name="T">用于数组存储的类型参数</typeparam>
public class FixedSizeCollection<T>
{
    /// <summary>
    /// 构造函数,增加静态计数器,设置内部存储
    /// </summary>
    /// <param name="items"></param>
    public FixedSizeCollection(int items)
    {
        FixedSizeCollection<T>.InstanceCount++;
        this.Items = new T[items];
    }

    /// <summary>
    /// 将一个数据项添加到类中,该数据项的类型由实例化的类型参数确定
    /// </summary>
    /// <param name="item">要添加的数据项</param>
    /// <returns>新添加的数据项的从0开始的索引</returns>
    public int AddItem(T item)
    {
        if (this.ItemCount < this.Items.Length)
        {
            this.Items[this.ItemCount] = item;
            return this.ItemCount++;
        }
        else
```

```
            throw new Exception("Item queue is full");
    }

    /// <summary>
    /// 从类中获得一个数据项
    /// </summary>
    /// <param name="index">要获取的数据项的从0开始的索引</param>
    /// <returns>实例化的类型的一个数据项</returns>
    public T GetItem(int index)
    {
        if (index >= this.Items.Length &&
            index >= 0)
            throw new ArgumentOutOfRangeException(nameof(index));

        return this.Items[index];
    }

    #region Properties
    /// <summary>
    /// 静态的实例计数器，用于泛型类实例化的类型
    /// </summary>
    public static int InstanceCount { get; set; }

    /// <summary>
    /// 类中包含的数据项的数量
    /// </summary>
    public int ItemCount { get; private set; }

    /// <summary>
    /// 类中包含的数据项
    /// </summary>
    private T[] Items { get; set; }
    #endregion // Properties

    /// <summary>
    /// 重写ToString以提供类的详细信息
    /// </summary>
    /// <returns>包含类详细信息、格式化过的字符串</returns>
    public override string ToString() =>
                $"There are {FixedSizeCollection<T>.InstanceCount.ToString()}
                instances of {this.GetType().ToString()} and this instance
                contains {this.ItemCount} items...";
}
```

当你查看 Items 数组属性的实现时，FixedSizeCollection<T> 开始变得有些不同了。Items 数组声明如下：

```
private T[] Items { get; set; }
```

而不是：

```
private object[] Items { get; set; }
```

Items 数组属性使用泛型类的类型参数（<T>）来确定允许哪些类型的数据项。FixedSizeCollection 用 object 作为 Items 数组属性的类型，它允许把任何类型存储在数

据项的数组中（因为所有类型都可转换为 object）；而 FixedSizeCollection<T> 通过类型参数确定允许哪些类型的对象，从而提供了类型安全。另外要注意的是，这些属性没有声明相关联的私有字段以存储数组。这个示例使用了 C# 3.0 中新增的*自动实现的属性*。在底层，C# 编译器为属性对应的类型创建了一个存储元素，但是只要你不需要在访问属性时执行特定的代码，就不再需要为此属性存储编写代码。要使属性只读，只要将 set; 声明标记为 private 即可。

在 AddItem 和 GetItem 的方法声明中可以看出另一个区别。AddItem 现在需要类型为 T 的一个参数，而在 FixedSizeCollection 中，它需要一个 object 类型的参数。GetItem 现在返回一个类型为 T 的值，而在 FixedSizeCollection 中，它返回一个 object 类型的值。这些改变允许 FixedSizeCollection<T> 中的方法使用实例化的类型存储和获取数组中的数据项，而不必像在 FixedSizeCollection 中那样允许存储任何 object。

```
/// <summary>
/// 将一个数据项添加到类中,该数据项的类型由实例化的类型参数确定
/// </summary>
/// <param name="item">要添加的数据项</param>
/// <returns>新添加的数据项的从0开始的索引</returns>
public int AddItem(T item)
{
    if (this.ItemCount < this.Items.Length)
    {
        this.Items[this.ItemCount] = item;
        return this.ItemCount++;
    }
    else
        throw new Exception("Item queue is full");
}

/// <summary>
/// 从类中获得一个数据项
/// </summary>
/// <param name="index">要获取的数据项的从0开始的索引</param>
/// <returns>实例化的类型的一个数据项</returns>
public T GetItem(int index)
{

    if (index >= this.Items.Length &&
        index >= 0)
        throw new ArgumentOutOfRangeException("index");

    return this.Items[index];
}
```

这提供了几个优势。首先最重要的是，FixedSizeCollection<T> 为数组中的数据项提供的类型安全。可以在 FixedSizeCollection 中编写如下代码：

```
// 常规类
FixedSizeCollection C = new FixedSizeCollection(5);
Console.WriteLine(C);

string s1 = "s1";
```

```
string s2 = "s2";
string s3 = "s3";
int i1 = 1;

// 添加到固定大小的集合中(作为object)
C.AddItem(s1);
C.AddItem(s2);
C.AddItem(s3);
// 将一个int添加到string数组中,完全没问题
C.AddItem(i1);
```

但是如果你尝试执行相同的操作，FixedSizeCollection<T> 将返回一个错误给编译器。

```
// 泛型类
FixedSizeCollection<string> gC = new FixedSizeCollection<string>(5);
Console.WriteLine(gC);

string s1 = "s1";
string s2 = "s2";
string s3 = "s3";
int i1 = 1;
// 添加到泛型类(作为string)
gC.AddItem(s1);
gC.AddItem(s2);
gC.AddItem(s3);
// 试图将一个int添加到string实例,被编译器拒绝
// error CS1503: Argument '1': cannot convert from 'int' to 'string'
//gC.AddItem(i1);
```

由编译器阻止它在运行时导致错误是个非常好的想法。

也许你并不会立刻注意到，但是在 FixedSizeCollection 中把整数添加到 object 数组中时，实际会对整数进行装箱。在 FixedSizeCollection 上调用 GetItem 的 IL 中可以看出这一点。

```
IL_0177: ldloc.2
IL_0178: ldloc.s i1
IL_017a: box [mscorlib]System.Int32
IL_017f: callvirt instance int32
       CSharpRecipes.ClassesAndGenerics/FixedSizeCollection::AddItem(object)
```

这一装箱操作把值类型的 int 转变成引用类型（object），以便存储在数组中。这导致在 object 数组中保存值类型时需要额外的工作。

在 FixedSizeCollection 的实现中从类取回一个数据项时会遇到另一个问题。看一下 FixedSizeCollection.GetItem 如何获取一个数据项。

```
// 保存获取的string
string sHolder;

// 需要进行转换,否则会出现错误CS0266:
// 无法隐式地将类型object转换为string
sHolder = (string)C.GetItem(1);
```

由于 FixedSizeCollection.GetItem 返回的数据项是 object 类型，需要将其强制转换成 string，以便获得你所希望的用于索引 1 位置的 string。它可能并不是一个 string，你只

能确定它是一个 object，但是必须将它强制转换成一种更具体的类型，才可以正确地给它赋值。

FixedSizeCollection<T> 的实现修正了这些问题。与 FixedSizeCollection 不同，FixedSizeCollection<T> 中并不需要拆箱操作，因为 GetItem 的返回类型是实例化的类型，编译器通过检查将要返回的值确保了这一点。

```
// 保存获取的string
string sHolder;
int iHolder;

// 不需要类型转换
sHolder = gC.GetItem(1);

// 试图将一个string保存到int变量
// 错误CS0029：无法隐式地将类型'string'转换为'int'
//iHolder = gC.GetItem(1);
```

为了看出两种类型之间的另一个区别，分别实例化每种类型的几个实例，代码如下所示。

```
// 常规类
FixedSizeCollection A = new FixedSizeCollection(5);
Console.WriteLine(A);
FixedSizeCollection B = new FixedSizeCollection(5);
Console.WriteLine(B);
FixedSizeCollection C = new FixedSizeCollection(5);
Console.WriteLine(C);

// 泛型类
FixedSizeCollection<bool> gA = new FixedSizeCollection<bool>(5);
Console.WriteLine(gA);
FixedSizeCollection<int> gB = new FixedSizeCollection<int>(5);
Console.WriteLine(gB);
FixedSizeCollection<string> gC = new FixedSizeCollection<string>(5);
Console.WriteLine(gC);
FixedSizeCollection<string> gD = new FixedSizeCollection<string>(5);
Console.WriteLine(gD);
```

上述代码的输出结果如下所示。

```
There are 1 instances of CSharpRecipes.ClassesAndGenerics+FixedSizeCollection
    and this instance contains 0 items...
There are 2 instances of CSharpRecipes.ClassesAndGenerics+FixedSizeCollection
    and this instance contains 0 items...
There are 3 instances of CSharpRecipes.ClassesAndGenerics+FixedSizeCollection
    and this instance contains 0 items...
There are 1 instances of CSharpRecipes.ClassesAndGenerics+FixedSizeCollection'1
    [System.Boolean] and this instance contains 0 items...
There are 1 instances of CSharpRecipes.ClassesAndGenerics+FixedSizeCollection'1
    [System.Int32] and this instance contains 0 items...
There are 1 instances of CSharpRecipes.ClassesAndGenerics+FixedSizeCollection'1
    [System.String] and this instance contains 0 items...
There are 2 instances of CSharpRecipes.ClassesAndGenerics+FixedSizeCollection'1
    [System.String] and this instance contains 0 items...
```

1.10.3 讨论

即使你不知道将要处理的最终类型，泛型中的类型参数也能让你创建类型安全的代码。在许多实例中，你希望类型具有某些特征，在这种情况下可以对类型施加一些限制（参见范例 1.12，即 1.12 节）。方法可以具有泛型类型参数，而不管类自身是否具有它们。

注意常规类 `FixedSizeCollection` 具有三个实例，而泛型类 `FixedSizeCollection<T>` 有一个声明为 `bool` 类型的实例，一个声明为 `int` 的实例，两个声明为 `string` 类型的实例。这意味着每个非泛型类对应创建了一个 .NET `Type` 对象，而一个泛型类的每个类型实例化都创建了一个 .NET `Type` 对象。

在示例代码中，`FixedSizeCollection` 具有三个实例，因为 `FixedSizeCollection` 只有一个由 CLR 维护的类型。对于泛型，每个类模板与类型实例构造时传入的类型参数的组合都维护了一个类型。换句话说，你得到了一个用于 `FixedSizeCollection<bool>` 的 .NET 类型，一个用于 `FixedSizeCollection<int>` 的 .NET 类型，还有一个用于 `FixedSizeCollection<string>` 的 .NET 类型。

静态属性 `InstanceCount` 有助于阐释这一点，因为类的静态属性实际上与 CLR 维护的类型相关联。CLR 只会对任何给定的类型创建一次，然后维护它，直到应用程序域卸载。这也就是发生以下情况的原因：对这些对象调用 `ToString()` 的结果显示 `FixedSizeCollection` 的计数是 `3`（因为确实只有其中 1 个计数器），而 `FixedSizeCollection<T>` 类型的计数器是 `1` 或 `2`。

1.10.4 参考

MSDN 文档中的“泛型类型参数”和“泛型类”主题。

1.11 反转有序列表中的内容

1.11.1 问题

你希望能够反转数据项的有序列表中的内容，并且还能够同时以数组和列表方式访问它们，就像使用 `SortedList` 和泛型类 `SortedList<T>` 那样。`SortedList` 和 `SortedList<T>` 都没有提供除重加载列表之外完成该任务的直接方式。

1.11.2 解决方案

使用 LINQ to Object 查询 `SortedList<T>`，并对列表中的信息应用降序排序。在实例化一个键为 `int`、值为 `string` 的 `SortedList<TKey, TValue>` 之后，把一系列无序的数字以及它们的文本表示插入列表中。然后会显示这些数据项。

```
SortedList<int, string> data = new SortedList<int, string>()
    { [2]="two", [5]="five", [3]="three", [1]="one" };

foreach (KeyValuePair<int, string> kvp in data)
```

```
{
    Console.WriteLine($"\t {kvp.Key}\t{kvp.Value}");
}
```

然后以升序（默认情况）排序的方式显示列表的输出。

```
1    one
2    two
3    three
5    five
```

现在，通过使用 LINQ to Object 创建一个查询并把 `orderby` 子句设置为 `descending` 来反转排序顺序。然后从查询结果集中显示结果：

```
// 降序排序查询
var query = from d in data
            orderby d.Key descending
            select d;

foreach (KeyValuePair<int, string> kvp in query)
{
    Console.WriteLine($"\t {kvp.Key}\t{kvp.Value}");
}
```

这一次结果以降序排序显示：

```
5    five
3    three
2    two
1    one
```

当把新的数据项添加到列表中时，将以升序的排序方式添加它，但是通过在添加了所有数据项之后再次查询列表，可以使列表的排序方式保持不变。

```
data.Add(4, "four");

// 降序排序重新查询
query = from d in data
        orderby d.Key descending
        select d;

foreach (KeyValuePair<int, string> kvp in query)
{
    Console.WriteLine($"\t {kvp.Key}\t{kvp.Value}");
}
Console.WriteLine("");

// 访问原始列表以升序方式显示
foreach (KeyValuePair<int, string> kvp in data)
{
    Console.WriteLine($"\t {kvp.Key}\t{kvp.Value}");
}
```

然后可以在以升序或降序排序的输出中看到新添加的数据项：

```
5    five
4    four
3    three
2    two
1    one

1    one
2    two
3    three
4    four
5    five
```

1.11.3　讨论

SortedList 融合了数组和列表的语法，允许以任何一种方式来访问数据，用起来很方便。可以以键值对或者直接通过索引访问数据，并且不允许添加重复的键。此外，引用或可空类型的值可以是 null，但是键不能是 null。可以使用 foreach 循环迭代访问数据项，其返回类型是 KeyValuePair。在访问 SortedList<T> 的元素时，只能读取它们。通常的迭代器语法禁止在读取列表中的元素时更新或删除它们，因为这将使迭代器无效。

查询子句中的 orderby 子句将导致查询的结果集以升序（默认）或降序排序。这种排序是使用针对元素类型的默认比较器完成的，因此对于自定义类型的元素，可以通过重写其 Equals 方法来影响它。可以为 orderby 子句指定多个键，因为它具有嵌套排序的作用，比如先按“last name”排序，再按“first name”排序。

1.11.4　参考

MSDN 文档中的“SortedList”“泛型 KeyValuePair 结构”和“泛型 SortedList”主题。

1.12　约束类型参数

1.12.1　问题

你需要用一种类型参数来创建泛型类型，该类型参数必须支持特定接口的成员，如 IDisposable。

1.12.2　解决方案

使用约束条件强制要求泛型类型的类型参数是一种实现一个或多个特定接口的类型。

```
public class DisposableList<T> : IList<T>
    where T : class, IDisposable
{
    private List<T> _items = new List<T>();

    // 私有方法,处置列表中的元素
    private void Delete(T item) => item.Dispose();
```

```
// IList<T> Members
public int IndexOf(T item) => _items.IndexOf(item);

public void Insert(int index, T item) => _items.Insert(index, item);

public T this[int index]
{
    get    {return (_items[index]);}
    set    {_items[index] = value;}
}

public void RemoveAt(int index)
{
    Delete(this[index]);
    _items.RemoveAt(index);
}

// ICollection<T> Members
public void Add(T item) => _items.Add(item);

public bool Contains(T item) => _items.Contains(item);

public void CopyTo(T[] array, int arrayIndex) =>
                 _items.CopyTo(array, arrayIndex);

public int Count  => _items.Count;

public bool IsReadOnly  => false;

// IEnumerable<T> Members
public IEnumerator<T> GetEnumerator()=> _items.GetEnumerator();

// IEnumerable Members
IEnumerator IEnumerable.GetEnumerator()=> _items.GetEnumerator();

// Other members
public void Clear()
{
    for (int index = 0; index < _items.Count; index++)
    {
        Delete(_items[index]);
    }

    _items.Clear();
}

public bool Remove(T item)
{
    int index = _items.IndexOf(item);

    if (index >= 0)
    {
        Delete(_items[index]);
        _items.RemoveAt(index);
```

```
                return (true);
            }
            else
            {
                return (false);
            }
        }
    }
```

这个 DisposableList 类只允许用一个实现了 IDisposable 接口的对象作为类型参数传入。其原因是，无论何时从 DisposableList 对象中删除一个对象，都会在该对象上调用 Dispose 方法。这允许你透明地管理这个 DisposableList 对象中存储的任何对象。

下面的代码用到了 DisposableList 对象：

```
    public static void TestDisposableListCls()
    {
        DisposableList<StreamReader> dl = new DisposableList<StreamReader>();

         // 创建一些测试对象
         StreamReader tr1 = new StreamReader("C:\\Windows\\system.ini");
         StreamReader tr2 = new StreamReader("c:\\Windows\\vmgcoinstall.log");
         StreamReader tr3 = new StreamReader("c:\\Windows\\Starter.xml");

         // 将测试对象添加到DisposableList
         dl.Add(tr1);
         dl.Insert(0, tr2);
         dl.Add(tr3);

         foreach(StreamReader sr in dl)
         {
             Console.WriteLine($"sr.ReadLine() == {sr.ReadLine()}");
         }

        // 在从DisposableList中移除任何可处置的对象之前调用Dispose方法
        dl.RemoveAt(0);
        dl.Remove(tr1);
        dl.Clear();
    }
```

1.12.3 讨论

where 关键字是用于约束类型参数只接受满足给定约束条件的参数。例如，DisposableList 约束任何类型参数 T 必须实现 IDisposable 接口。

```
    public class DisposableList<T> : IList<T>
            where T : IDisposable
```

这意味着下面的代码将成功地编译：

```
    DisposableList<StreamReader> dl = new DisposableList<StreamReader>();
```

但是下面的代码则不然：

```
DisposableList<string> dl = new DisposableList<string>();
```

这是由于 `string` 类型没有实现 `IDisposable` 接口，而 `StreamReader` 类型则实现了这个接口。

除了要求实现一个或多个特定的接口之外，还允许对类型参数施加其他约束条件。可以强制类型参数继承自特定的基类（如 `TextReader` 类）。

```
public class DisposableList<T> : IList<T>
        where T : System.IO.TextReader, IDisposable
```

还可以确定是否把类型参数限制为只能是值类型或引用类型。下面的类声明被限制为只使用值类型。

```
public class DisposableList<T> : IList<T>
          where T : struct
```

下面这个类声明被限制为只使用引用类型。

```
public class DisposableList<T> : IList<T>
          where T : class
```

此外，还可以要求类型参数实现公开的默认构造函数。

```
public class DisposableList<T> : IList<T>
        where T : IDisposable, new()
```

使用约束条件允许编写只接受一组小范围可用类型参数的泛型类型。如果 1.12.2 节中省略了 `IDisposable` 约束条件，将会发生编译时错误。这是由于并非所有的类型都可用作将实现 `IDisposable` 接口的 `DisposableList` 类的类型参数。如果跳过这个编译时检查，`DisposableList` 对象就可能包含没有公开的无参 `Dispose` 方法的对象。在这种情况下，就会发生运行时异常。泛型，尤其是约束条件可以强制对类型的参数执行严格的类型检查，并且允许在编译时而非运行时捕获这些问题。

1.12.4　参考

MSDN 文档中的“`where` 关键字”主题。

1.13　将泛型变量初始化为默认值

1.13.1　问题

你有一个泛型类，它包含一个变量，其类型与类自身定义的类型参数的类型相同。在构造泛型对象时，你希望将该变量初始化为它的默认值。

1.13.2　解决方案

简单地使用 `default` 关键字将该变量初始化为它的默认值。

```
public class DefaultValueExample<T>
{
    T data = default(T);

    public bool IsDefaultData()
    {
        T temp = default(T);

        if (temp.Equals(data))
        {
            return (true);
        }
        else
        {
            return (false);
        }
    }

    public void SetData(T val) => data = value;
}
```

使用这个类的代码如下所示。

```
public static void ShowSettingFieldsToDefaults()
{
    DefaultValueExample<int> dv = new DefaultValueExample<int>();

    // 检查是否将数据设置为其默认值;返回true
    bool isDefault = dv.IsDefaultData();
    Console.WriteLine($"Initial data: {isDefault}");
    // 设置数据
    dv.SetData(100);
    // 再次检查,这次返回的是false
    isDefault = dv.IsDefaultData();
    Console.WriteLine($"Set data: {isDefault}");
}
```

第一次调用 `IsDefaultData` 将返回 `true`，而第二次调用则会返回 `false`。输出如下所示。

```
Initial data: True
Set data: False
```

1.13.3 讨论

在初始化与泛型类相同的类型参数的变量时，不能仅仅把该变量设置为 `null`。如果类型参数是一种值类型（如 `int` 或 `char`）会发生什么情况呢？这是行不通的，因为值类型不能为 `null`。你可能认为可以把可空类型（如 `long?` 或 `Nullable<long>`）设置为 `null`（参考 MSDN 文档中“使用可以为 `null` 的类型（C# 编程指南）”了解可空类型的更多信息）。不过，编译器无法知道用户将使用什么类型参数来构造类型。

`default` 关键字允许在编译时告诉编译器应该使用这个变量的默认值。如果提供的类型参数是一个数值（例如 `int`、`long`、`decimal`），那么默认值将是 `0`。如果提供的类型参数是一种引用类型，那么默认值将是 `null`。如果提供的类型参数是一个 `struct`，那么通过把每个

成员字段初始化为它们的默认值来确定该 struct 的默认值。

1.13.4 参考

MSDN 文档中的“使用可以为 null 的类型（C# 编程指南）”和“泛型代码中的默认关键字”主题。

1.14 向生成的实体中添加钩子

1.14.1 问题

你有一种生成分部类业务实体定义的过程，并且想添加一种轻量级的通知机制。

1.14.2 解决方案

使用分部方法在生成的代码中为业务实体添加钩子。

生成实体的过程可能来自于 UML、数据集或者其他对象建模工具，但是当代码被生成为分部类时，为调用 ChangingProperty 分部方法的属性会将分部方法挂钩添加到模板中，如 GeneratedEntity 类中所示。

```
public partial class GeneratedEntity
{
    public GeneratedEntity(string entityName)
    {
        this.EntityName = entityName;
    }

    partial void ChangingProperty(string name, string originalValue,
                                  string newValue);

    public string EntityName { get; }
    private string _FirstName;
    public string FirstName
    {
        get { return _FirstName; }
        set
        {
            ChangingProperty("FirstName",_FirstName,value);
            _FirstName = value;
        }
    }
    private string _State;
    public string State
    {
        get { return _State; }
        set
        {
            ChangingProperty("State",_State,value);
            _State = value;
```

```
        }
    }
}
```

GeneratedEntity 具有两个属性：FirstName 和 State。注意其中每个属性都具有相同的样板代码，它使用属性的名称、原始值和新值调用 ChangingProperty 方法。如果此时使用生成的类，编译器将删除 ChangingProperty 的声明和方法，因为不存在 ChangingProperty 的实现。如果像下面这样提供一个实现来报告属性变化，那么将保留并执行 ChangingProperty 的所有分部方法代码：

```
public partial class GeneratedEntity
{
    partial void ChangingProperty(string name, string originalValue,
                                  string newValue)
    {
        Console.WriteLine($"Changed property ({name}) for entity " +
                          $"{this.EntityName} from " +
                          $"{originalValue} to {newValue}");
    }
}
```

1.14.3 讨论

在使用分部方法时，要注意以下几点。

- 用 partial 修饰符指示分部方法。
- 只能在分部类中声明分部方法。
- 分部方法可以只有方法声明，而没有方法体。
- 从签名角度讲，分部方法可以包含参数，需要空返回值，不能有任何访问修饰符。分部意味着它是私有方法，并且可以是静态、泛型或者不安全的。
- 对于泛型分部方法，约束条件必须在声明和实现上重复标记。
- 分部方法不能实现接口成员，因为接口成员必须是公共的。
- 不能使用 virtual、abstract、override、new、sealed 或 extern 这些修饰符。
- 分部方法的参数不能使用 out，但可以使用 ref。

分部方法类似于条件方法，只不过条件方法中总会包含方法定义，甚至当条件未满足时也是如此。如果没有匹配的实现，分部方法将不会保留方法定义。可以像下面这样使用 1.14.2 节中的代码。

```
public static void TestPartialMethods()
{
    Console.WriteLine("Start entity work");
    GeneratedEntity entity = new GeneratedEntity("FirstEntity");
    entity.FirstName = "Bob";
    entity.State = "NH";
    GeneratedEntity secondEntity = new GeneratedEntity("SecondEntity");
    entity.FirstName = "Jay";
    secondEntity.FirstName = "Steve";
    secondEntity.State = "MA";
    entity.FirstName = "Barry";
```

```
        secondEntity.State = "WA";
        secondEntity.FirstName = "Matt";
        Console.WriteLine("End entity work");
    }
```

在提供了 ChangingProperty 的实现时，将产生以下输出。

```
Start entity work
Changed property (FirstName) for entity FirstEntity from to Bob
Changed property (State) for entity FirstEntity from to NH
Changed property (FirstName) for entity FirstEntity from Bob to Jay
Changed property (FirstName) for entity SecondEntity from to Steve
Changed property (State) for entity SecondEntity from to MA
Changed property (FirstName) for entity FirstEntity from Jay to Barry
Changed property (State) for entity SecondEntity from MA to WA
Changed property (FirstName) for entity SecondEntity from Steve to Matt
End entity work
```

在没有提供 ChangingProperty 的实现时，将产生以下输出。

```
Start entity work
End entity work
```

1.14.4 参考

MSDN 文档中的“分部方法”和“分部（方法）”主题。

1.15 控制如何触发多播委托中的一个委托

1.15.1 问题

你组合了多个委托来创建一个多播委托。当调用这个多播委托时，将依次调用其中的每个委托。你需要施加更多的控制。例如，调用每个委托的顺序，只触发一个委托子集或者基于前一个委托成功与否来触发每个委托。此外，你需要能够单独处理每个委托的返回值。

1.15.2 解决方案

使用 GetInvocationList 方法获得 Delegate 对象的数组。接下来，使用 for 循环（如果以非标准顺序进行枚举）或 foreach 循环（如果以标准顺序进行枚举）遍历这个数组。然后可以逐个调用数组中的每个 Delegate 对象，并且可以获取每个委托的返回值。

在 C# 中，所有委托类型都支持多播；也就是说，如果这样设置，那么任何委托实例在每次调用时都可以调用多个方法。在本范例中，我们使用术语*多播*（multicast）来描述设置为调用多个方法的委托。

下面的方法创建一个名为 allInstances 的多播委托，然后使用 GetInvocationList 以逆序逐一调用每个委托。Func<int> 泛型委托用于创建返回 int 的委托实例。

```
public static void InvokeInReverse()
{
```

```
    Func<int> myDelegateInstance1 = TestInvokeIntReturn.Method1;
    Func<int> myDelegateInstance2 = TestInvokeIntReturn.Method2;
    Func<int> myDelegateInstance3 = TestInvokeIntReturn.Method3;

    Func<int> allInstances =
            myDelegateInstance1 +
            myDelegateInstance2 +
            myDelegateInstance3;

    Console.WriteLine("Fire delegates in reverse");
    Delegate[] delegateList = allInstances.GetInvocationList();
    foreach (Func<int> instance in delegateList.Reverse())
    {
        instance();
    }
}
```

注意，为了翻转使用 GetInvocationList 得到的委托列表，我们使用了 IEnumerable<T> 的扩展方法 Reverse，以枚举通常产生数据项时的相反顺序获得它们。

如下面的方法所示，通过触发每隔一个的委托，你不必调用列表中的所有委托。InvokeEveryOtherOperation 使用此处为 IEnumerable<T> 创建的名为 EveryOther 的扩展方法，该方法从枚举中每隔一个数据项进行返回。

> 如果使用单播委托，并且对它调用 GetInvocationList，将会获得一个包含单一委托实例的列表。

```
public static void InvokeEveryOtherOperation()
{
    Func<int> myDelegateInstance1 = TestInvokeIntReturn.Method1;
    Func<int> myDelegateInstance2 = TestInvokeIntReturn.Method2;
    Func<int> myDelegateInstance3 = TestInvokeIntReturn.Method3;

    Func<int> allInstances = myDelegateInstance1 +
                             myDelegateInstance2 +
                             myDelegateInstance3;

    Delegate[] delegateList = allInstances.GetInvocationList();
    Console.WriteLine("Invoke every other delegate");
    foreach (Func<int> instance in delegateList.EveryOther())
    {
        // 调用委托
        int retVal = instance();
        Console.WriteLine($"Delegate returned {retVal}");
    }
}

static IEnumerable<T> EveryOther<T>(this IEnumerable<T> enumerable)
{
    bool retNext = true;
```

```
        foreach (T t in enumerable)
        {
            if (retNext) yield return t;
            retNext = !retNext;
        }
    }
```

下面的类包含被多播委托 allInstances 调用的所有方法。

```
    public class TestInvokeIntReturn
    {
        public static int Method1()
        {
            Console.WriteLine("Invoked Method1");
            return 1;
        }

        public static int Method2()
        {
            Console.WriteLine("Invoked Method2");
            return 2;
        }

        public static int Method3()
        {
            Console.WriteLine("Invoked Method3");
            return 3;
        }
    }
```

还可以基于当前触发委托的返回值来决定是否继续触发列表中的委托。下面的方法将会触发每个委托，仅当委托返回 false 时才会停止。

```
    public static void InvokeWithTest()
    {
        Func<bool> myDelegateInstanceBool1 = TestInvokeBoolReturn.Method1;
        Func<bool> myDelegateInstanceBool2 = TestInvokeBoolReturn.Method2;
        Func<bool> myDelegateInstanceBool3 = TestInvokeBoolReturn.Method3;

        Func<bool> allInstancesBool =
                myDelegateInstanceBool1 +
                myDelegateInstanceBool2 +
                myDelegateInstanceBool3;

        Console.WriteLine(
            "Invoke individually (Call based on previous return value):");
        foreach (Func<bool> instance in allInstancesBool.GetInvocationList())
        {
            if (!instance())
                break;
        }
    }
```

下面的类包含被多播委托 allInstancesBool 调用的所有方法。

```
public class TestInvokeBoolReturn
{
    public static bool Method1()
    {
        Console.WriteLine("Invoked Method1");
        return true;
    }

    public static bool Method2()
    {
        Console.WriteLine("Invoked Method2");
        return false;
    }

    public static bool Method3()
    {
        Console.WriteLine("Invoked Method3");
        return true;
    }
}
```

1.15.3 讨论

当一个委托被调用时，它将调用其*调用列表*内存储的所有委托。这些委托通常是按添加顺序逐一调用的。使用 MulticastDelegate 类的 GetInvocationList 方法，可以获得多播委托的调用列表中的每个委托。该方法不接受任何参数，并且返回 Delegate 对象的数组，对应于调用此方法的委托对象的调用列表。返回的 Delegate 数组包含调用列表中的委托，其排列顺序是通常调用它们的顺序；也就是说，Delegate 数组中的第一个元素包含一般情况下最先调用的 Delegate 对象。

GetInvocationList 方法的这种应用可让你精确控制何时以及如何调用多播委托中的委托，并且在一个委托失败时允许你阻止继续调用委托。如果每个委托都在操作数据，并且其中一个委托在履行其职责时失败但是没有引发异常，那么这种能力就很重要了。如果一个委托在履行其职责时失败，并且其余的委托依赖前面所有的委托来成功履行其职责，就必须在失败时停止调用委托。

此范例更加高效地处理了委托失败，还在处理这些错误时提供了更多的灵活性。例如，可以编写逻辑以基于之前调用委托的返回值来指定要调用哪些委托。下面的方法调用一个名为 All 的多播委托，然后使用 GetInvocationList 单独触发每个委托。在触发每个委托后，捕获其返回值。

```
public static void TestIndividualInvokesReturnValue()
{
    Func<int> myDelegateInstance1 = TestInvokeIntReturn.Method1;
    Func<int> myDelegateInstance2 = TestInvokeIntReturn.Method2;
    Func<int> myDelegateInstance3 = TestInvokeIntReturn.Method3;

    Func<int> allInstances =
            myDelegateInstance1 +
            myDelegateInstance2 +
            myDelegateInstance3;
```

```
        Console.WriteLine("Invoke individually (Obtain each return value):");
        foreach (Func<int> instance in allInstances.GetInvocationList())
        {
            int retVal = instance();
            Console.WriteLine($"\tOutput: {retVal}");
        }
    }
```

多播委托的一个怪异之处是，如果其调用列表内的任意或所有委托返回了一个值，那么只会返回最后一个所调用委托的返回值，所有其他值都会丢失。这可能会令人苦恼；如果代码需要这些返回值，事情会变得更糟。考虑像通常那样调用 allInstances 委托的情况，如下面的代码所示。

```
retVal = allInstances();
Console.WriteLine(retVal);
```

这将会显示值 3，因为 Method3 是 allInstances 委托调用的最后一个方法。所有其他返回值都不会被捕获。

通过使用 MulticastDelegate 类的 GetInvocationList 方法，可以绕过这种限制。该方法返回 Delegate 对象的数组，其中每个对象都可以单独进行调用。注意，该方法不会调用每个委托，而是只会把它们的数组返回给调用者。通过单独调用每个委托，可以从每个调用的委托获取每个返回值。

注意在调用多播委托时，所有 out 或 ref 参数都会丢失。此范例使得你可以获得多播委托内每个调用委托的 out 和 / 或 ref 参数。

尽管如此，你仍然需要意识到，从其中一个调用委托中发生的任何未处理异常都会冒泡到本范例中展示的 TestIndividualInvokesReturnValue 方法。如果在多播委托内调用的委托中发生异常并且该异常未处理，就不会调用任何余下的委托。这是多播委托预期的行为。不过，在某些情况下，你希望能够处理各个委托中引发的异常，然后在那一刻决定是否继续调用余下的委托。

> 异常将强制停止调用委托。异常应该仅用于异常情况，不应该用于控制流程。

在如下的 TestIndividualInvokesExceptions 方法中，如果捕获到一个异常，就会把它记录到事件日志中并显示它，然后代码继续调用委托。

```
public static void TestIndividualInvokesExceptions()
{
    Func<int> myDelegateInstance1 = TestInvokeIntReturn.Method1;
    Func<int> myDelegateInstance2 = TestInvokeIntReturn.Method2;
    Func<int> myDelegateInstance3 = TestInvokeIntReturn.Method3;

    Func<int> allInstances =
            myDelegateInstance1 +
            myDelegateInstance2 +
```

```
            myDelegateInstance3;

    Console.WriteLine("Invoke individually (handle exceptions):");

    // 创建一个封装异常的实例以包含委托实例调用中遇到的任何异常
    List<Exception> invocationExceptions = new List<Exception>();

    foreach (Func<int> instance in allInstances.GetInvocationList())
    {
        try
        {
            int retVal = instance();
            Console.WriteLine($"\tOutput: {retVal}");
        }
        catch (Exception ex)
        {
            // 显示并记录异常,然后继续执行
            Console.WriteLine(ex.ToString());
            EventLog myLog = new EventLog();
            myLog.Source = "MyApplicationSource";
            myLog.WriteEntry(
                $"Failure invoking {instance.Method.Name} with error " +
                $"{ex.ToString()}",
                EventLogEntryType.Error);
            // 将此异常添加到列表
            invocationExceptions.Add(ex);
        }
    }
    // 如果捕获了任何异常,引发包含所有异常的封装异常
    if (invocationExceptions.Count > 0)
    {
        throw new MulticastInvocationException(invocationExceptions);
    }
}
```


可以向上述的 MulticastInvocationException 类中添加多个异常。它通过 InvocationExceptions 属性公开一个 ReadOnlyCollection<Exception>，如下所示。

```
[Serializable]
public class MulticastInvocationException : Exception
{
    private List<Exception> _invocationExceptions;

    public MulticastInvocationException()
        : base()
    {
    }

    public MulticastInvocationException(
    IEnumerable<Exception> invocationExceptions)
    {
        _invocationExceptions = new List<Exception>(invocationExceptions);
    }

    public MulticastInvocationException(string message)
```

```
            : base(message)
        {
        }

        public MulticastInvocationException(string message, Exception innerException)
            :base(message,innerException)
        {
        }

        protected MulticastInvocationException(SerializationInfo info,
            StreamingContext
                   context) :
            base(info, context)
        {
            _invocationExceptions =
                (List<Exception>)info.GetValue("InvocationExceptions",
                    typeof(List<Exception>));
        }

        [SecurityPermissionAttribute(SecurityAction.Demand,
            SerializationFormatter = true)]
        public override void GetObjectData(
           SerializationInfo info, StreamingContext context)
        {
            info.AddValue("InvocationExceptions", this.InvocationExceptions);
            base.GetObjectData(info, context);
        }

        public ReadOnlyCollection<Exception> InvocationExceptions =>
                    new ReadOnlyCollection<Exception>(_invocationExceptions);
    }
```

这种策略允许根据需要对异常进行细粒度的处理。一种选择是把在委托处理期间发生的所有异常都存储起来，然后把处理期间遇到的所有异常包装在一个自定义的异常中。在处理完成之后，引发这个自定义的异常。

通过向这个 `try-catch` 块中添加一个 `finally` 块，可以确保在每个委托返回后执行这个 `finally` 块内的代码。如果想在调用委托的代码之间穿插一些代码（比如用于清理不需要对象的代码，或者用于验证每个委托用到的数据是否保持稳定状态的代码），这种技术就很有用。

1.15.4 参考

MSDN 文档中的“`Delegate` 类”和“`Delegate.GetInvocationList` 方法”主题。

1.16 在C#中使用闭包

1.16.1 问题

你希望把少量状态与某种行为关联起来，但不会陷入构建新类的麻烦之中。

1.16.2 解决方案

使用 lambda 表达式实现闭包。闭包是声明时捕获作用域中环境状态的函数。简单地讲，它们是当前状态以及某种可以读取和修改该状态的行为。lambda 表达式能够捕获外部变量并延长它们的生存期，这使得闭包可以在 C# 中使用。

关于 lambda 表达式的更多信息，请参考 4.0 节。

作为闭包的一个示例，我们将构建一个快速报告系统，用于跟踪销售人员及其收益和佣金。闭包的行为是，你可以构建一些代码，用于计算每个季度的佣金并且用于每个销售人员。

首先，必须定义销售人员，代码如下所示。

```
class SalesPerson
{
    // CTOR的
    public SalesPerson()
    {
    }

    public SalesPerson(string name,
                       decimal annualQuota,
                       decimal commissionRate)
    {
        this.Name = name;
        this.AnnualQuota = annualQuota;
        this.CommissionRate = commissionRate;
    }

    // 私有成员
    decimal _commission;

    // 属性
    public string Name { get; set; }

    public decimal AnnualQuota { get; set; }

    public decimal CommissionRate { get; set; }

    public decimal Commission
    {
        get { return _commission; }
        set
        {
            _commission = value;
            this.TotalCommission += _commission;
        }
    }
```

```
        public decimal TotalCommission {get; private set; }
    }
```

销售人员有姓名、年度定额、销售佣金率，以及用于存储每季度佣金和总佣金的属性。既然有一些事情要做，就让我们编写一些代码来计算佣金。

```
    delegate void CalculateEarnings(SalesPerson sp);

    static CalculateEarnings GetEarningsCalculator(decimal quarterlySales,
                                     decimal bonusRate)
    {
         return salesPerson =>
         {
            // 计算salesPerson的季度指标
            decimal quarterlyQuota = (salesPerson.AnnualQuota / 4);
            // 他达成季度指标了吗
            if (quarterlySales < quarterlyQuota)
            {
                // 未达成指标,没有佣金
                salesPerson.Commission = 0;
            }
            // 检查奖金级别的绩效(指标的200%)
            else if (quarterlySales > (quarterlyQuota * 2.0m))
            {
                decimal baseCommission = quarterlyQuota *
                salesPerson.CommissionRate;
                salesPerson.Commission = (baseCommission +
                        ((quarterlySales - quarterlyQuota) *
                        (salesPerson.CommissionRate * (1 + bonusRate))));
            }
            else // 常规的佣金
            {
                salesPerson.Commission =
                   salesPerson.CommissionRate * quarterlySales;
            }
        };
    }
```

将委托类型声明为 CalculationEarnings，它接受一个 SalesPerson 类型参数。有一个名为 GetEarningsCalculator 的工厂方法，用于构造此委托类型的一个实例。它将创建一个 lambda 表达式来计算 SalesPerson 的佣金，并返回一个 CalculateEarnings 的实例。

在开始前，必须创建 salespeople 数组，代码如下所示。

```
    // 设定salespeople……
    SalesPerson[] salesPeople = {
        new SalesPerson { Name="Chas", AnnualQuota=100000m, CommissionRate=0.10m },
        new SalesPerson { Name="Ray", AnnualQuota=200000m, CommissionRate=0.025m },
        new SalesPerson { Name="Biff", AnnualQuota=50000m, CommissionRate=0.001m }};
```

然后基于每季度的收入建立收入计算器，代码如下所示。

```
    public class QuarterlyEarning
    {
        public string Name { get; set; }
```

```
        public decimal Earnings { get; set; }
        public decimal Rate { get; set; }
    }
    QuarterlyEarning[] quarterlyEarnings =
                { new QuarterlyEarning(){ Name="Q1", Earnings = 65000m, Rate = 0.1m },
                  new QuarterlyEarning(){ Name="Q2", Earnings = 20000m, Rate = 0.1m },
                  new QuarterlyEarning(){ Name="Q3", Earnings = 37000m, Rate = 0.1m },
                  new QuarterlyEarning(){ Name="Q4", Earnings = 110000m, Rate = 0.15m}
                };

    var calculators = from e in quarterlyEarnings
                      select new
                      {
                          Calculator =
                              GetEarningsCalculator(e.Earnings, e.Rate),
                          QuarterlyEarning = e
                      };
```

最后，统计每个季度的所有 salespeople 数值，然后通过该数据调用 WriteCommission-Report 生成年度报告。这将告诉主管人员哪些销售人员值得留下。

```
    decimal annualEarnings = 0;
    foreach (var c in calculators)
    {
        WriteQuarterlyReport(c.QuarterlyEarning.Name,
            c.QuarterlyEarning.Earnings, c.Calculator, salesPeople);
        annualEarnings += c.QuarterlyEarning.Earnings;
    }

    // 看一下谁值得留下
    WriteCommissionReport(annualEarnings, salesPeople);
```

WriteQuarterlyReport 为每个 SalesPerson 调用 CalculateEarnings 的 lambda 表达式实现（eCalc），并且基于每个销售人员的佣金率来修改状态以对每季度的佣金进行赋值。

```
    static void WriteQuarterlyReport(string quarter,
                                decimal quarterlySales,
                                CalculateEarnings eCalc,
                                SalesPerson[] salesPeople)
    {
        Console.WriteLine($"{quarter} Sales Earnings on Quarterly Sales of
                            {quarterlySales.ToString("C")}:");
        foreach (SalesPerson salesPerson in salesPeople)
        {
            // 计算佣金
            eCalc(salesPerson);
            // 报告
            Console.WriteLine($"\tSales person {salesPerson.Name} " +
                               "made a commission of : " +
                              $"{salesPerson.Commission.ToString("C")}");
        }
    }
```

WriteCommissionReport 对比检查各个销售人员的佣金及其所实现的收益。如果佣金超过了

其产生收益的 20%，就要采取建议的动作。

```
static void WriteCommissionReport(decimal annualEarnings,
                                SalesPerson[] salesPeople)
{
    decimal revenueProduced = ((annualEarnings) / salesPeople.Length);
    Console.WriteLine("");

    Console.WriteLine($"Annual Earnings were {annualEarnings.ToString("C")}");
    Console.WriteLine("");
    var whoToCan = from salesPerson in salesPeople
                   select new
                   {
                       // 如果佣金超过了他产生收入的20%,解雇他
                       CanThem = (revenueProduced * 0.2m) <
                                   salesPerson.TotalCommission,
                       salesPerson.Name,
                       salesPerson.TotalCommission
                   };

    foreach (var salesPersonInfo in whoToCan)
    {
        Console.WriteLine($"\t\tPaid {salesPersonInfo.Name} " +
                 $"{salesPersonInfo.TotalCommission.ToString("C")} to produce" +
                 $"{revenueProduced.ToString("C")}");
        if (salesPersonInfo.CanThem)
        {
            Console.WriteLine($"\t\t\tFIRE {salesPersonInfo.Name}!");
        }
    }
}
```

下面列出了收益和佣金跟踪程序的输出。

```
Q1 Sales Earnings on Quarterly Sales of $65,000.00:
         SalesPerson Chas made a commission of : $6,900.00
         SalesPerson Ray made a commission of : $1,625.00
         SalesPerson Biff made a commission of : $70.25
Q2 Sales Earnings on Quarterly Sales of $20,000.00:
         SalesPerson Chas made a commission of : $0.00
         SalesPerson Ray made a commission of : $0.00
         SalesPerson Biff made a commission of : $20.00
Q3 Sales Earnings on Quarterly Sales of $37,000.00:
         SalesPerson Chas made a commission of : $3,700.00
         SalesPerson Ray made a commission of : $0.00
         SalesPerson Biff made a commission of : $39.45
Q4 Sales Earnings on Quarterly Sales of $110,000.00:
         SalesPerson Chas made a commission of : $12,275.00
         SalesPerson Ray made a commission of : $2,975.00
         SalesPerson Biff made a commission of : $124.63

Annual Earnings were $232,000.00

    Paid Chas $22,875.00 to produce $77,333.33
        FIRE Chas!
```

```
    Paid Ray $4,600.00 to produce $77,333.33
    Paid Biff $254.33 to produce $77,333.33
```

1.16.3　讨论

对 C# 中闭包的最佳描述之一是把对象视为与数据关联的一组方法，并把闭包视为与一个函数关联的一组数据。如果需要对相同的数据执行多种不同的操作，使用对象可能更有意义一些。它们是处理相同问题的两种不同角度，要解决的问题类型有助于决定用哪个方法更合适。它只依赖你倾向于选择哪种方法。有时候，百分之百纯面向对象编程可能是冗长乏味而且不必要的，可以使用闭包很好地解决其中一些问题。这里展示的 `SalesPerson` 佣金示例演示了可以利用闭包做什么。不使用它们也可以完成任务，但是其代价是要编写更多的类和方法代码。

之前对闭包已经进行了定义，但是有一种更严格的定义：它实质上意味着与状态关联的行为不应该能够修改状态，以便使之成为真正的闭包。我们倾向于赞同第一种定义，因为它表达了闭包应该是什么，而不是闭包应该如何实现，后者的限制性过强。无论你选择将其视为 lambda 表达式某个方面的优雅特性，还是觉得值得将它称作闭包，它都是工具箱中的又一种编程技巧，不应该被摒弃。

1.16.4　参考

范例 1.17（即 1.17 节）和 MSDN 文档中的“lambda 表达式”主题。

1.17　使用函数对象在列表中执行多种操作

1.17.1　问题

你希望能够同时对整个对象集合执行多种操作，并在功能上隔离这些操作。

1.17.2　解决方案

使用函数对象（functor 或 function object）作为转换集合的工具。函数对象是任何一个可以作为函数被调用的对象。例如，委托、函数、函数指针，甚至是 C/C++ 中定义了 `operator()` 的对象。

在软件中，经常需要对一个集合执行多种操作。假定你的股票组合包含了一系列股票。`StockPortfolio` 类包含一个 `Stock` 对象的 `List`，并且能够添加股票。

```
public class StockPortfolio : IEnumerable<Stock>
{
    List<Stock> _stocks;

    public StockPortfolio()
    {
        _stocks = new List<Stock>();
    }
```

```
    public void Add(string ticker, double gainLoss)
    {
        _stocks.Add(new Stock() {Ticker=ticker, GainLoss=gainLoss});
    }

    public IEnumerable<Stock> GetWorstPerformers(int topNumber) =>
          _stocks.OrderBy((Stock stock) => stock.GainLoss).Take(topNumber);

    public void SellStocks(IEnumerable<Stock> stocks)
    {
        foreach(Stock s in stocks)
            _stocks.Remove(s);
    }

    public void PrintPortfolio(string title)
    {
        Console.WriteLine(title);
        _stocks.DisplayStocks();
    }

    #region IEnumerable<Stock> Members
    public IEnumerator<Stock> GetEnumerator() => _stocks.GetEnumerator();
    #endregion

    #region IEnumerable Members
    IEnumerator IEnumerable.GetEnumerator() => this.GetEnumerator();
    #endregion
}
```

Stock 类相当简单。只需要一个股票代码及其利润或亏损百分比。

```
public class Stock
{
    public double GainLoss { get; set; }
    public string Ticker { get; set; }
}
```

要使用这个 StockPortfolio，可以向其中添加几支带利润 / 亏损百分比的股票，并输出初始股票组合。有了这个股票组合，你希望得到三支表现最差股票的列表，以便可以通过卖出股票来改进股票组合，然后再次输出它。

```
StockPortfolio tech = new StockPortfolio() {
    {"OU81", -10.5},
    {"C#6VR", 2.0},
    {"PCKD", 12.3},
    {"BTML", 0.5},
    {"NOVB", -35.2},
    {"MGDCD", 15.7},
    {"GNRCS", 4.0},
    {"FNCTR", 9.16},
    {"LMBDA", 9.12},
    {"PCLS", 6.11}};

tech.PrintPortfolio("Starting Portfolio");
```

```
// 出售表现最差的3支股票
var worstPerformers = tech.GetWorstPerformers(3);
Console.WriteLine("Selling the worst performers:");
worstPerformers.DisplayStocks();

tech.SellStocks(worstPerformers);
tech.PrintPortfolio("After Selling Worst 3 Performers");
```

迄今为止，没有发生任何特别令人感兴趣的事情。通过查看 GetWorstPerformers 方法的内部代码，看一下如何查明三支最差的股票。

```
public IEnumerable<Stock> GetWorstPerformers(int topNumber) => _stocks.OrderBy(
                        (Stock stock) => stock.GainLoss).Take(topNumber);
```

首先通过调用 IEnumerable<T> 的 OrderBy 扩展方法确保列表有序，以便将表现最差的股票列在列表的前面。OrderBy 方法接受一个 lambda 表达式，它提供了用于比较的利润 / 亏损百分比，以找出 Take 扩展方法中 topNumber 指示的股票数量。

GetWorstPerformers 返回一个 IEnumerable<Stock>，包含三支表现最差的股票。既然它们没有赚到钱，你应该兑现并卖出它们。对你来说，卖出股票只是简单地在 StockPortfolio 中从股票列表中删除它们。要实现这一点，可使用另一个函数对象来遍历提交给 SellStocks 函数的股票列表（这里是指表现最差的股票列表），然后从 StockPortfolio 类维护的内部列表中删除该股票。

```
public void SellStocks(IEnumerable<Stock> stocks)
{
    foreach(Stock s in stocks)
        _stocks.Remove(s);
}
```

1.17.3 讨论

函数对象具有几种不同的样式：*生成器*（不带参数的函数）、*一元函数*（带一个参数的函数）和*二元函数*（带两个参数的函数）。如果函数对象恰好返回一个布尔值，那么它就有一个更为特定的命名约定：返回布尔值的一元函数称为谓词；返回布尔值的二元函数称为二元谓词。在 Framework 中包含了 Predicate<T> 和 BinaryPredicate<T> 的定义以便于应用这些函数对象。

List<T> 和 System.Array 类接受谓词（Predicatec<T>、BinaryPredicate<T>）、动作（Action<T>）、比较（Comparison<T>）和转换（Converter<T,U>）。这允许以比之前更通用的方式来操作这些集合。

最初以函数对象的方式来思考会有一些挑战，但是一旦花点时间研究它，就会开始看到它带来的强大可能性。任何能够编写一次、调试一次，然后多次使用的代码都是有价值的，函数对象能帮助你达到这一点。

上述示例的输出如下所示。

```
Starting Portfolio
  (OU81) lost 10.5%
```

```
  (C#6VR) gained 2%
  (PCKD) gained 12.3%
  (BTML) gained 0.5%
  (NOVB) lost 35.2%
  (MGDCD) gained 15.7%
  (GNRCS) gained 4%
  (FNCTR) gained 9.16%
  (LMBDA) gained 9.12%
  (PCLS) gained 6.11%
Selling the worst performers:
  (NOVB) lost 35.2%
  (OU81) lost 10.5%
  (BTML) gained 0.5%
After Selling Worst 3 Performers
  (C#6VR) gained 2%
  (PCKD) gained 12.3%
  (MGDCD) gained 15.7%
  (GNRCS) gained 4%
  (FNCTR) gained 9.16%
  (LMBDA) gained 9.12%
  (PCLS) gained 6.11%
```

1.17.4　参考

MSDN 文档中的“System.Collections.Generic.List<T>”“System.Linq.Enumerable 类”和“System.Array”主题。

1.18　控制结构类型字段初始化

1.18.1　问题

你需要能够控制结构的初始化，取决于是否想要将结构的所有内部字段初始化为基于字段类型的标准默认值（例如，int 初始化为 0，string 初始化为空字符串），或者初始化为一组非标准的默认值，或者初始化为一组预定义的值。

1.18.2　解决方案

可以使用结构的各种构造函数来实现我们的目标。要将结构的所有内部字段初始化为基于字段类型的标准默认值，只需使用结构的默认初始化即可，稍后会演示这一点。要将结构的字段初始化为一组预定义的值，可以使用重载的构造函数。最后，要将结构初始化为一组非标准的默认值，需要在结构构造函数中使用可选参数。通过可选参数，结果能够将其内部字段设置为构造函数参数列表中可选参数指定的默认值。

例 1-8 中的数据结构使用了重载的构造函数初始化结构的所有字段。

例 1-8：带有重载构造函数的结构

```
public struct Data
{
```

```
    public Data(int intData, float floatData, string strData,
                char charData, bool boolData)
    {
        IntData = intData;
        FloatData = floatData;
        StrData = strData;
        CharData = charData;
        BoolData = boolData;
    }

    public int IntData { get; }
    public float FloatData { get; }
    public string StrData { get; }
    public char CharData { get; }
    public bool BoolData { get; }

    public override string ToString()=> IntData + " :: " + FloatData + " :: " +
                       StrData + " :: " + CharData + " :: " + BoolData;
}
```

这是初始化结构字段值的典型方式。同时要注意，存在一个隐式的默认构造函数允许结构将其字段初始化为字段的默认值。不过，你也许想要将每个字段初始化为非默认值。例 1-9 中的数据结构使用重载的、带有可选参数的构造函数，将结构的所有字段初始化为非默认值。

例 1-9：带有包含可选参数的构造函数的结构

```
public struct Data
{
    public Data(int intData, float floatData = 1.1f, string strData = "a",
                char charData = 'a', bool boolData = true) : this()
    {
        IntData = intData;
        FloatData = floatData;
        StrData = strData;
        CharData = charData;
        BoolData = boolData;
    }

    public int IntData { get; }
    public float FloatData { get; }
    public string StrData { get; }
    public char CharData { get; }
    public bool BoolData { get; }

    public override string ToString()=> IntData + " :: " + FloatData + " :: " +
                       StrData + " :: " + CharData + " :: " + BoolData;
}
```

当然，可以引入一个新的初始化方法来使得事情更为简单。但是你需要显式调用它，如例 1-10 所示。

例 1-10：带有显式初始化方法的结构

```
public struct Data
{
```

```
        public void Init()
        {
            IntData = 2;
            FloatData = 1.1f;
            StrData = "AA";
            CharData = 'A';
            BoolData = true;
        }

        public int IntData { get; private set; }
        public float FloatData { get; private set; }
        public string StrData { get; private set; }
        public char CharData { get; private set; }
        public bool BoolData { get; private set; }

        public override string ToString()=> IntData + " :: " + FloatData + " :: " +
                           StrData + " :: " + CharData + " :: " + BoolData;
    }
```

注意，当使用类似 Init 这样的显式初始化方法时，需要为每个属性添加私有的属性 setter 来将每个字段初始化。

1.18.3　讨论

我们能够以不同的技巧创建例 1-8 所示结构的实例。每种技巧都使用了初始化此结构对象的不同方法。第一种技巧使用了 default 关键字来创建这个结构，代码如下所示。

```
Data dat = default(Data);
```

default 关键字只是创建了这个结构的一个实例，并将它的所有字段初始化为字段的默认值。本质上，所有的数值类型默认为 0，bool 类型默认为 false，char 默认为 '\0'，string 和其他引用类型默认为 null。

现在，如果你并不介意引用类型和 char 设置为 null 值，那就太好了；但是假设你需要在创建结构时将这些类型设置为除 null 之外的值呢？第二种技巧正是用于解决这个问题的；它使用默认无参构造函数，代码如下所示。

```
Data dat = new Data();
```

这一代码使得默认无参构造函数被调用。要附带说明的是，在使用结构的默认无参构造函数时，必须使用 new 关键字创建结构的一个实例。如果没有 new 关键字，默认构造函数是不会被调用的。因此，下面的代码并没有调用到默认无参构造函数。

```
Data[] dat = new Data[4];
```

相反，这会用到该结构每个字段的系统定义默认值。

有两种方法可以解决这个问题。你可以使用冗长的方式创建 Data 结构的一个数组，代码为：

```
Data[] dat = new Data[4];

dat[0] = new Data();
dat[1] = new Data();
```

```
dat[2] = new Data();
dat[3] = new Data();
```

或者：

```
ArrayList dat = new ArrayList();
dat.Add(new Data());
dat.Add(new Data());
dat.Add(new Data());
dat.Add(new Data());
```

你也可以使用简洁的选择，其中用到了 LINQ。

```
Data[] dataList = new Data[4];
dataList = (from d in dataList
            select new Data()).ToArray();
```

LINQ 表达式迭代整个 `Data` 数组，为每个 `Data` 类型的结构元素显式调用了默认无参构造函数。

如果前两种选择都无法用于你的特定案例，你总是可以创建一个重载的构造函数，为每个想要初始化的字段传入参数。第三种技巧需要使用重载的构造函数来创建此结构的一个新实例，代码如下所示。

```
public Data(int intData, float floatData, string strData,
            char charData, bool boolData)
{
    IntData = intData;
    FloatData = floatData;
    StrData = strData;
    CharData = charData;
    BoolData = boolData;
}
```

这个构造函数显式地将每个字段初始化为用户提供的值。

```
Data dat = new Data(2, 2.2f, "blank", 'a', false);
```

在 C# 6.0 中，你不仅可以选择将结构的字段初始化为系统默认值，或者使用重载的构造函数将字段初始化为用户定义的值，还可以使用带有可选参数的重载构造函数将字段初始化为非系统的默认值，如例 1-9 中所示。带有可选参数的构造函数看起来如下所示。

```
public Data(int intData, float floatData = 1.1f, string strData = "a",
            char charData = 'a', bool boolData = true) : this()
{
    ...
}
```

使用这种构造函数的一个问题是，你必须至少给这个构造函数提供一个参数值。如果 `intData` 参数也有一个关联的可选参数：

```
public Data(int intData = 2, float floatData = 1.1f, string strData = "a",
            char charData = 'a', bool boolData = true) : this()
{
```

```
        ...
    }
```

那么，下面的代码：

```
Data dat = new Data();
```

将调用结构的默认无参构造函数，而不是重载的构造函数。这是必须将至少一个参数传入这个构造函数的原因：

```
Data dat = new Data(3);
```

现在我们调用了重载的构造函数，将其第一个参数 `intData` 的值设为 3，并将其他参数设置为它们的可选值。

最后一个选择是，你可以将一个显式的初始化方法添加到结构中，用于将字段初始化为非默认值。这一技巧展示在例 1-10 中。

将 `Init` 方法添加到结构中后，必须在使用 `new` 或 `default` 关键字初始化结构之后调用它。接着 `Init` 方法将每个字段初始化为非默认值。其他需要做的代码修改仅仅是为结构的属性添加私有的 `setter` 方法。这使得 `Init` 方法可以设置内部字段而无需将它们暴露出来。

1.18.4　参考

MSDN 文档中的“结构”主题。

1.19　以更简洁的方式检查null值

1.19.1　问题

你不断地编写笨拙的 `if-then` 语句来判断一个对象是否为 `null`。你需要一种更简洁、更简单的方式来编写这类代码。

1.19.2　解决方案

使用 C# 6.0 中新引入的 `null` 条件运算符。在过去，通常需要在使用对象前进行检查以确保对象不为 `null`。

```
if (val != null)
{
    val.Trim().ToUpper();
    ...
}
```

现在你可以简单地使用 `null` 条件运算符。

```
val?.Trim().ToUpper();
```

这一简化的语法判断 `val` 是否为 `null`；如果是，那么将不会调用 `Trim` 和 `ToUpper` 方法，也就不会引发讨厌的 `NullReferenceException`。如果 `val` 不是 `null`，则将调用 `Trim` 和

ToUpper 方法。

在使用点运算符将一系列对象成员访问链到一起时，也可以使用 null 条件运算符测试每个对象是否为 null。

```
Person?.Address?.State?.Trim();
```

在这种情况下，如果前面三个对象（Person、Address 和 State）中任何一个为 null，点运算符将不再对 null 对象进行调用，此表达式的执行也会终止。

null 条件运算符不仅可以用于常规对象，也可以用于数组和索引以及返回的索引元素。例如，如果 val 的类型是 string[]，这行代码将检查 val 变量的值是否为 null。

```
val?[0].ToUpper();
```

然而下面这行代码检查保存在 val 数组中 0 索引位置上的实际 string 元素是否为 null。

```
val[0]?.ToUpper();
```

下面这一行代码也是有效的；它同时检查 val 和 0 索引的元素是否为 null。

```
val?[0]?.ToUpper();
```

null 条件运算符表现突出的另外一个领域是调用委托和事件。例如，如果你有一个简单的委托：

```
public delegate bool Approval();
```

并且使用 lambda 表达化实例化它，为了简化而直接返回 true：

```
Approval approvalDelegate = () => { return true; };
```

之后当你想调用这一委托时，就不需要编写任何笨重的条件代码去判断委托是否为 null；只要简单地使用 null 条件运算符即可。

```
approvalDelegate?.Invoke()
```

1.19.3 讨论

本质上，null 条件运算符以类似于三元运算符（?:）的方式工作。以下代码：

```
val?.Trim();
```

是以下代码的简写：

```
(val != null ) ? (string)val.Trim() : null
```

上述代码假设 val 的类型是 string。

让我们来看一下如果返回的是值类型会发生什么，代码如下所示。

```
val?.Length;
```

上述表达式被修改为返回一个可空的值类型，例如 int?。

```
(val != null ) ? (int?)val.Length : null
```

这意味着你不能简单地使用 null 运算符并将返回值赋给任意类型 ，它必须是一个可空类型。因此，下面这行代码是无法编译的。

```
int len = val?.Length;
```

但是下面这行代码可以编译。

```
int? len = val?.Length;
```

注意，仅在值类型的情况下才需要将返回类型变成可空类型。

此外，你不能在期望一个非空类型的地方尝试使用 null 条件运算符。例如，数组的大小期望一个 int 值，所以你不能编译下面这行代码。

```
byte[] data = new byte[val?.Length];
```

不过，你可以使用 GetValueOrDefault 方法将可空类型的值转化为非空类型友好的值，代码如下所示。

```
byte[] data = new byte[(val?.Length).GetValueOrDefault()];
```

这一方式下，如果 val 为 null，byte 数组将被初始化为整数类型的默认值，也就是 0。注意这一方法会返回值类型的默认值，对于数值类型为 0，对于 bool 类型为 false。你的代码必须考虑到这一点，以保证应用程序行为的一致性。在这个例子中，当 val 对象的长度为 0 或者 val 为 null 时，byte 数组的大小为 0，因此你的应用程序逻辑必须处理这一点。

需要在条件语句中使用这一运算符时也要小心。

```
if (val?.Length > 0)
    Console.WriteLine("val.length > 0");
else
    Console.WriteLine("val.length = 0 or null");
```

在这个条件语句中，如果 val 变量不为 null 并且长度大于 0，if 语句的真值语句块将会执行并显示文本 "val.length > 0"。如果 val 为 null，假值语句块将执行并显示文本 "val.length = 0 or null"。然而，你并不知道 val 究竟是什么：是 null 还是 0 呢？

如果需要检查 val 的长度是否为 0，可以在 if-else 语句中添加额外的检查，以考虑所有情况。

```
if (val?.Length > 0)
    Console.WriteLine("val.Length > 0");
else if (val?.Length == 0)
    Console.WriteLine("val.Length = 0");
else
    Console.WriteLine("val.Length = null");
```

switch 语句以如下相似的方式运行。

```
switch (val?.Length)
{
    case 0:
```

```
            Console.WriteLine("val.Length = 0");
            break;
        case 1:
            Console.WriteLine("val.Length = 1");
            break;
        default:
            Console.WriteLine("val.Length > 1 or val.Length = null");
            break;
    }
```

如果 `val` 为 `null`，将跳转到 `default` 语句块执行。除非进行更多的检查，你无法知道 `val` 的长度是大于 1 还是 `null`。

在条件语句中使用 `null` 条件运算符时要小心。如果不够小心，这一用法可能导致代码中的逻辑错误。

1.19.4　参考

MSDN 文档中的“`Null` 条件运算符”主题。

第 2 章

集合、枚举器和迭代器

2.0 简介

集合是一组数据项；在 .NET 中，集合包含对象，而包含在集合中的每个对象被称为*元素*（element）。有些集合包含简单的元素列表，而另外一些集合［比如*字典*（dictionary）］则包含键值对的列表。下列集合类型包含简单的元素列表。

```
System.Collections.ArrayList
System.Collections.BitArray
System.Collections.Queue
System.Collections.Stack
System.Collections.Generic.LinkedList<T>
System.Collections.Generic.List<T>
System.Collections.Generic.Queue<T>
System.Collections.Generic.Stack<T>
System.Collections.Generic.HashSet<T>
```

下列集合类型都是字典。

```
System.Collections.Hashtable
System.Collections.SortedList
System.Collections.Generic.Dictionary<T,U>
System.Collections.Generic.SortedList<T,U>
```

最后一种集合类型（`HashSet<T>`）可被视为无重复的元素列表。

```
System.Collections.Generic.HashSet<T>
```

这些集合类都组织在 `System.Collections` 和 `System.Collections.Generic` 命名空间下。除了这些命名空间之外，另一个名为 `System.Collections.Specialized` 的命名空间中还包

含另外几个有用的集合类。这些类不像上述那些类那样为人熟知，下面给出了它们的简要解释。

- ListDictionary

  该类的操作方式类似于 Hashtable。不过，当包含 10 个或 10 个以下的元素时，该类的性能要优于 Hashtable。

- HybridDictionary

  该类包含两个内部集合：ListDictionary 和 Hashtable。在任一时刻只会使用其中一个类。当集合中包含 10 个或 10 个以下的元素时，将使用 ListDictionary，而当集合中包含的元素增长到 10 个以上时，就切换到使用 Hashtable。这种切换对于开发人员是透明的。一旦使用 Hashtable，该集合就不能回复到使用 ListDictionary，即使元素个数降到 10 个以下时也是如此。另外要注意，当使用字符串作为键时，该类通过在构造函数中设置一个布尔值，可以同时支持区分大小写（考虑到固定区域性）和不区分大小写的字符串查找。

- CollectionsUtil

  该类包含两个静态方法：一个用于创建不区分大小写的 Hashtable，另一个用于创建不区分大小写的 SortedList。在直接创建 Hashtable 和 SortedList 对象时，总会创建区分大小写的 Hashtable 或 SortedList，除非你使用某个接受 IComparer 参数的构造函数，并将 CaseInsensitiveComparer.Default 传递给它。

- NameValueCollection

  这个集合包含键值对，其中键和值都是 String 类型。关于这个集合，有一件有趣的事情是，它可以用一个键存储多个字符串值。多个字符串之间用逗号隔开。在隔开一个值中的多个字符串时，可以使用 String.Split 方法。

- StringCollection

  这个集合是包含字符串元素的简单列表。该列表接受 null 元素以及重复的字符串。该列表区分大小写。

- StringDictionary

  这是一个 Hashtable，它将键和值存储为字符串。在把键添加到 Hashtable 中之前，将它们全部转换成小写字母，从而允许进行不区分大小写的比较。键不能为 null，但是值可以设置成 null。

C# 编译器还支持固定大小的数组。可以使用以下语法创建任意类型的数组。

```
int[] foo = new int[2];
T[] bar = new T[2];
```

这里，foo 是一个整型数组，其中正好包含两个元素，bar 是未知类型 T 的一个数组。

数组也可以具有多种样式，比如单维数组、锯齿形数组，甚至锯齿形多维数组。多维数组的定义如下所示。

```
int[,] foo = new int[2,3];      // 一个包含6个元素的二维数组
int[,,] bar = new int[2,3,4];   // 一个包含24个元素的三维数组
```

通常将二维数组描述为具有行和列的表。foo 数组可被描述为一个包括两行的表，其中每一行包含三个元素列。三维数组可以被描述为一个有多层表的立方体。bar 数组可以被描述为 4 层，每一层包含两行，每一行包含三个元素列。

锯齿形数组是数组的数组。如果把锯齿形数组描述成一个一维数组，该数组中的每个元素包含另一个一维数组，那么每一行中就可以具有不同的元素个数。锯齿形数组的定义如下所示。

```
int[][] baz = new int[2][] {new int[2], new int[3]};
```

baz 数组包含一个一维数组，该数组包含两个元素。其中每个元素都包含另一个数组，第一个数组具有两个元素，第二个数组具有三个元素。

在使用集合时，某些时候你可能需要检查集合中的所有值。为了帮助你做到这一点，C# 提供了迭代器和枚举器构造。*迭代器*（iterator）允许代码块产生有序的值，而*枚举器*（enumerator）支持迭代数据集，并且可以用于读取集合中的数据，但是不能修改它。

迭代器是一种机制，用于产生可以被 foreach 循环构造遍历的数据。不过，迭代器要比这灵活得多。你可以轻松生成由枚举器返回的数据序列［称为*迟缓计算*（lazy computation）］，而不必预先进行硬编码［如同*积极计算*（eager computation）中所做的］。例如，你可以根据需要轻松地编写一个生成斐波那契数列的枚举器。迭代器的另一个灵活的特性是，不必对迭代器返回的值的个数设置限制。因此在本示例中，可以选择何时停止产生斐波那契数列。这是 LINQ 世界中一个有趣的特点。IEnumerable 的 where 查询生成的迭代器是迟缓的，而分组或排序需要积极计算。

迭代器允许你把编写这个类的工作移交给 C# 编译器。现在，你只需把迭代器添加到类型中。迭代器是类型中的一个成员（例如，方法、运算符重载或属性的 get 访问器），该成员返回一个 System.Collections.IEnumerator、System.Collections.Generic.IEnumerator<T>、System.Collections.IEnumerable 或 System.Collections.Generic.IEnumerator<T>，并且包含至少一个 yield 语句。这允许你编写可以被 foreach 循环使用的类型。

迭代器在 LINQ 中起着重要的作用，因为 LINQ to Object 基于能够操作那些实现了 IEnumerable<T> 的类。迭代器允许查询引擎在遍历集合时执行多种不同的查询、投影、排序和分组操作。如果没有迭代器的支持，LINQ 将变得极其麻烦，并且它引入的声明性编程风格将变得很笨拙，甚至完全失去这种能力。

2.1　寻找List<T>中的重复数据项

2.1.1　问题

你需要能够对一个 List<T> 中匹配搜索条件的对象进行读取或计数操作。

2.1.2 解决方案

使用 List<T> 的四个扩展方法：GetAll、BinarySearchGetAll、CountAll 和 BinarySearchCountAll。这些方法扩展 List<T> 类，以返回特定的对象实例或特定对象出现在有序和无序 List<T> 中的次数，如例 2-1 所示。

例 2-1：确定一个数据项在 List<T> 中出现的次数

```
static class CollectionExtMethods
{
    #region 2.1 寻找List<T>中的重复数据项

    // 从一个有序或无序的List<T>中获取所有匹配对象的方法
    public static IEnumerable<T> GetAll<T>(this List<T> myList, T searchValue) =>
        myList.Where(t => t.Equals(searchValue));

    // 从一个有序的List<T>中获取所有匹配对象的方法
    public static T[] BinarySearchGetAll<T>(this List<T> myList, T searchValue)
    {
        List<T> retObjs = new List<T>();

        // 查找第一个元素
        int center = myList.BinarySearch(searchValue);
        if (center > 0)
        {
            retObjs.Add(myList[center]);

            int left = center;
            while (left > 0 && myList[left - 1].Equals(searchValue))
            {
                left -= 1;
                retObjs.Add(myList[left]);
            }

            int right = center;
            while (right < (myList.Count - 1) &&
                myList[right + 1].Equals(searchValue))
            {
                right += 1;
                retObjs.Add(myList[right]);
            }
        }

        return (retObjs.ToArray());
    }
    // 统计一个数据项在无序或有序的List<T>中出现的次数
    public static int CountAll<T>(this List<T> myList, T searchValue) =>
        myList.GetAll(searchValue).Count();

    // 统计一个数据项在有序的List<T>中出现的次数
    public static int BinarySearchCountAll<T>(this List<T> myList, T searchValue) =>
        BinarySearchGetAll(myList, searchValue).Count();
    #endregion // 2.1
}
```

2.1.3 讨论

GetAll 和 BinarySearchGetAll 方法返回在 List<T> 对象中找到的实际数据项。CountAll 和 BinarySearchCountAll 方法利用 GetAll 和 BinarySearchGetAll 以提供数据项的计数。在 GetAll 和 BinarySearchGetAll 之间进行选择时要牢记的主要事情是：将要查找的 List<T> 是否为有序的。选择 GetAll 和 CountAll 方法从无序的 List<T> 中获取所有查找到的数据项的数组（GetAll）或者查找到数据项的数量（CountAll），对于有序的 List<T> 则选择 BinarySearchAll 和 BinarySearchCountAll。GetAll、SearchAll 和 BinarySearchAll 使用了表达式 – 函数体成员语法，因为它们是简单函数。

下面的代码使用了 List<T> 类的这两个新的扩展方法。

```
// 获取
List<int> listRetrieval =
  new List<int>() { -1, -1, 1, 2, 2, 2, 2, 3, 100, 4, 5 };

Console.WriteLine("--GET All--");
IEnumerable<int> items = listRetrieval.GetAll(2);
foreach (var item in items)
    Console.WriteLine($"item: {item}");

Console.WriteLine();
items = listRetrieval.GetAll(-2);
foreach (var item in items)
    Console.WriteLine($"item-2: {item}");

Console.WriteLine();
items = listRetrieval.GetAll(5);
foreach (var item in items)
    Console.WriteLine($"item5: {item}");

Console.WriteLine("\r\n--BINARY SEARCH GET ALL--");
listRetrieval.Sort();
int[] listItems = listRetrieval.BinarySearchGetAll(-2);
foreach (var item in listItems)
    Console.WriteLine($"item-2: {item}");

Console.WriteLine();
listItems = listRetrieval.BinarySearchGetAll(2);
foreach (var item in listItems)
    Console.WriteLine($"item2: {item}");

Console.WriteLine();
listItems = listRetrieval.BinarySearchGetAll(5);
foreach (var item in listItems)
    Console.WriteLine($"item5: {item}");
```

这段代码的输出如下所示。

```
--GET All--
item: 2
item: 2
item: 2
```

```
item: 2


item5: 5

--BINARY SEARCH GET ALL--

item2: 2
item2: 2
item2: 2
item2: 2

item5: 5
```

BinarySearchGetAll 方法比 GetAll 方法快，尤其是当数组已排好序时。如果对无序的 List<T> 使用 BinarySearch，那么查找返回的结果将会是错误的，因为文档中一直将 List<T> 有序作为它的前提条件。

CountAll 方法接受一个泛型类型 T 的查找值（searchValue）。然后，该方法继续统计这个查找值在 List<T> 类中的出现次数，通过使用 GetAll 扩展方法获得数据项然后调用结果上的 Count。无论 List<T> 是否有序，都可以使用该方法。如果 List<T> 是有序的（通过调用 Sort 方法对 List<T> 排序），就可以使用 BinarySearchCountAll 方法提高查找效率。对 List<T> 类利用 BinarySearchGetAll 扩展方法来执行该任务，这比遍历整个 List<T> 快得多。当 List<T> 庞大时尤其如此。

下面的代码举例说明了 List<T> 类的两个新方法。

```
List<int> list = new List<int>() {-2,-2,-1,-1,1,2,2,2,2,3,100,4,5};

Console.WriteLine("--CONTAINS TOTAL--");
int count = list.CountAll(2);
Console.WriteLine($"Count2: {count}");

count = list.CountAll(3);
Console.WriteLine($"Count3: {count}");

count = list.CountAll(1);
Console.WriteLine($"Count1: {count}");

Console.WriteLine("\r\n--BINARY SEARCH COUNT ALL--");
list.Sort();
count = list.BinarySearchCountAll(2);
Console.WriteLine($"Count2: {count}");

count = list.BinarySearchCountAll(3);
Console.WriteLine($"Count3: {count}");

count = list.BinarySearchCountAll(1);
Console.WriteLine($"Count1: {count}");
```

这段代码的输出如下所示。

```
--CONTAINS TOTAL--
Count2: 4
Count3: 1
Count1: 1

--BINARY SEARCH COUNT ALL--
Count2: 4
Count3: 1
Count1: 1
```

CountAllt 和 GetAll 方法使用顺序查找，在一个 for 循环中执行。因为没有假定 List<T> 是有序的，所以必须使用线性查找。where 语句确定 List<T> 中的每个元素是否等于查找条件（searchValue）。这些方法返回数据项或数据项的计数以指示 List<T> 中与查找条件匹配的数据项个数。

BinarySearchGetAll 方法实现了一个二叉查找，用于定位 List<T> 中与查找条件（searchValue）匹配的数据项。如果找到这样一个数据项，就会使用一个 while 循环查找 List<T> 中第一个匹配的数据项，并把该元素的位置记录在 left 变量中。第二个 while 循环用于查找最后一个匹配的数据项，并把该元素的位置记录在 right 变量中。用 right 变量中的值减去 left 变量中的值，然后把得到的结果加上 1，就得到了总的匹配个数。BinarySearchCountAll 使用 BinarySearchGetAll 来获取数据项，然后仅对结果集调用 Count。

2.1.4　参考

MSDN 文档中的“List<T> 类”主题。

2.2　保持List<T>有序

2.2.1　问题

你将使用 List<T> 的 BinarySearch 方法定期查找 List<T> 中的特定元素。在查找过程中，将穿插进行添加、修改和删除元素的操作。不过，BinarySearch 方法预先假设数组是有序的；如果 List<T> 是无序的，BinarySearch 方法可能返回不正确的结果。你不希望必须记住在调用 List<T>.BinarySearch 方法之前调用 List<T>.Sort 方法，更不必说会引入与该调用关联的所有开销。你需要采用一种方法保持 List<T> 有序，而不必总是调用 List<T>.Sort 方法。

2.2.2　解决方案

下面的 SortedList 泛型类增强了在 List<T> 内添加和修改元素的操作。在向其添加数据项或修改数据项时，这些方法可以保持数组有序。注意，这里并不需要 DeleteSorted 方法，因为删除数据项不会干扰剩余数据项的排序顺序。

```
public class SortedList<T> : List<T>
{
```

```
    public new void Add(T item)
    {
        int position = this.BinarySearch(item);
        if (position < 0)
            position = ~position;

        this.Insert(position, item);
    }

    public void ModifySorted(T item, int index)
    {
        this.RemoveAt(index);

        int position = this.BinarySearch(item);
        if (position < 0)
            position = ~position;

        this.Insert(position, item);
    }
}
```

2.2.3 讨论

在保持 List<T> 有序的同时使用 Add 方法添加元素。Add 方法接受一个泛型类型（T）以添加到有序列表中。

不要直接使用 List<T> 索引器修改元素，而是使用 ModifySorted 方法修改元素，同时保持 List<T> 有序。调用这个方法，传入泛型类型 T 以替换现有的对象（item），并且传入要修改对象的索引（index）。

下面的代码演示了 SortedList<T> 类。

```
// 创建一个SortedList并用随机选择的数值填充
SortedList<int> sortedList = new SortedList<int>();
sortedList.Add(200);
sortedList.Add(20);
sortedList.Add(2);
sortedList.Add(7);
sortedList.Add(10);
sortedList.Add(0);
sortedList.Add(100);
sortedList.Add(-20);
sortedList.Add(56);
sortedList.Add(55);
sortedList.Add(57);
sortedList.Add(200);
sortedList.Add(-2);
sortedList.Add(-20);
sortedList.Add(55);
sortedList.Add(55);

// 显示列表
foreach (var i in sortedList)
    Console.WriteLine(i);
```

```
// 现在修改一些索引处的值
sortedList.ModifySorted(0, 5);
sortedList.ModifySorted(1, 10);
sortedList.ModifySorted(2, 11);
sortedList.ModifySorted(3, 7);
sortedList.ModifySorted(4, 2);
sortedList.ModifySorted(2, 4);
sortedList.ModifySorted(15, 0);
sortedList.ModifySorted(0, 15);
sortedList.ModifySorted(223, 15);

// 显示列表
Console.WriteLine();
foreach (var i in sortedList)
    Console.WriteLine(i);
```

该方法在保持 List<T> 排序顺序的同时自动把新数据项置于其中；执行该操作时，无需显式调用 List<T>.Sort。其原因是，Add 方法首先调用 BinarySearch 方法，并把要添加到 List<T> 中的对象传递给它。BinarySearch 方法将返回在其中找到相同数据项的索引或者返回一个负数，可以用返回值来确定查找的数据项应该位于什么位置。如果 BinarySearch 方法返回一个正数，就可以使用 List<T>.Insert 方法在那个位置插入一个新元素，保持 List<T> 内的排序顺序。如果 BinarySearch 方法返回一个负数，就可以使用按位求补运算符 ~ 来确定数据项应该位于什么位置，假定它存在于一个有序列表中。使用这个数字，可以使用 List<T>.Insert 方法把数据项添加到有序列表中的正确位置，同时保持正确的排序顺序。

可以在不干扰排序顺序的情况下从有序列表中删除元素，但是在 List<T> 中修改元素的值则极有可能导致有序列表变得无序。ModifySorted 方法缓解了这个问题。该方法的工作方式类似于 Add 方法，只不过它首先从 List<T> 中删除元素，然后把新元素插入正确的位置。

2.2.4　参考

MSDN 文档中的“List<T> 类”主题。

2.3　对Dictionary的键和/或值排序

2.3.1　问题

你想要对一个 Dictionary 中包含的键和 / 或值排序，以便把整个 Dictionary 显示给用户，并以升序或降序方式对它进行排序。

2.3.2　解决方案

通过 LINQ 查询以及 Dictionary<T,U> 对象的 Keys 和 Values 属性，获得其键和值对象的有序 ICollection。下面所示的代码显示以升序或降序排序的 Dictionary<T,U> 的键和值。

```
// 定义一个Dictionary<T,U>对象
Dictionary<string, string> hash = new Dictionary<string, string>()
{
    ["2"] = "two",
    ["1"] = "one",
    ["5"] = "five",
    ["4"] = "four",
    ["3"] = "three"
};

var x = from k in hash.Keys orderby k ascending select k;
foreach (string s in x)
    Console.WriteLine($"Key: {s}  Value: {hash[s]}");

x = from k in hash.Keys orderby k descending select k;
foreach (string s in x)
    Console.WriteLine($"Key: {s}  Value: {hash[s]}");
```

下面的代码显示以升序或降序排序的 Dictionary<T,U> 中的值。

```
x = from k in hash.Values orderby k ascending select k;
foreach (string s in x)
    Console.WriteLine($"Value: {s}");

Console.WriteLine();

x = from k in hash.Values orderby k descending select k;
foreach (string s in x)
    Console.WriteLine($"Value: {s}");
```

2.3.3 讨论

Dictionary<T,U> 对象公开了两个有用的属性，用于获得其键或值的集合。Keys 属性返回一个 ICollection，其中包含目前在 Dictionary<T,U> 中的所有键。Values 属性也返回一个 ICollection，其中包含目前在 Dictionary<T,U> 中的所有值。

通过 Dictionary<T,U> 对象的 Keys 或 Values 属性返回的 ICollection 对象包含对 Dictionary<T,U> 内的键和值集体的直接引用。这意味着如果 Dictionary<T,U> 中的键和 / 或值发生变化，键和值集合也将相应地改变。

注意，你还可以使用 SortedDictionary<T,U> 类，该类自动将键保持为有序。还可以使用 SortedDictionary<T,U> 的构造函数重载来包装现有的 Dictionary<T,U>。Keys 属性默认按升序排序，因此如果想要按降序排序，将需要基于 Keys 以降序方式对集合进行排序。

```
SortedDictionary<string, string> sortedHash =
    new SortedDictionary<string, string>()
{
    ["2"] = "two",
    ["1"] = "one",
    ["5"] = "five",
    ["4"] = "four",
    ["3"] = "three"
};
```

```
foreach (string key in sortedHash.Keys)
    Console.WriteLine($"Key: {key}  Value: {sortedHash[key]}");
foreach (string key in sortedHash.OrderByDescending(item =>
    item.Key).Select(item => item.Key))
    Console.WriteLine($"Key: {key}  Value: {sortedHash[key]}");
```

为什么有人会选择使用展示的 LINQ 方案，而非只是使用 SortedDictionary<T,U> 呢？在 LINQ 查询中执行排序实际上更快，代码也比使用 SortedDictionary<T,U> 更简洁，因此在支持 LINQ 的所有 .NET 版本（3.0 及更高）中这是推荐方法。如果你的解决方案碰巧是旧版本的 .NET，仍然可以使用 SortedDictionary<T,U> 来实现结果。

2.3.4 参考

MSDN 文档中的“Dictionary<T,U> 类”“SortedDictionary<T,U> 类”和“List<T> 类”主题。

2.4 创建具有最小值和最大值边界的Dictionary

2.4.1 问题

你需要在项目中使用一个泛型 Dictionary 对象，它在值中仅存储预定义最大值和最小值之间的数值数据（键可以是任意类型）。

2.4.2 解决方案

创建一个类，它带有强制执行这些边界的访问器和方法。例 2-2 中所示的类 MinMaxValueDictionary 只允许存储实现了 IComparable 接口的类型，并且其值位于最大值和最小值之间。

例 2-2：创建具有最小值和最大值边界的字典

```
[Serializable]
public class MinMaxValueDictionary<T, U>
    where U : IComparable<U>
{
    protected Dictionary<T, U> internalDictionary = null;

    public MinMaxValueDictionary(U minValue, U maxValue)
    {
        this.MinValue = minValue;
        this.MaxValue = maxValue;
        internalDictionary = new Dictionary<T, U>();
    }

    public U MinValue { get; private set; } = default(U);
    public U MaxValue { get; private set; } = default(U);

    public int Count => (internalDictionary.Count);

    public Dictionary<T, U>.KeyCollection Keys => (internalDictionary.Keys);
```

```
public Dictionary<T, U>.ValueCollection Values => (internalDictionary.Values);

public U this[T key]
{
    get { return (internalDictionary[key]); }
    set
    {
        if (value.CompareTo(MinValue) >= 0 &&
            value.CompareTo(MaxValue) <= 0)
            internalDictionary[key] = value;
        else
            throw new ArgumentOutOfRangeException(nameof(value), value,
                $"Value must be within the range {MinValue} to {MaxValue}");
    }
}

public void Add(T key, U value)
{
    if (value.CompareTo(MinValue) >= 0 &&
        value.CompareTo(MaxValue) <= 0)
        internalDictionary.Add(key, value);
    else
        throw new ArgumentOutOfRangeException(nameof(value), value,
            $"Value must be within the range {MinValue} to {MaxValue}");
}

public bool ContainsKey(T key) => (internalDictionary.ContainsKey(key));

public bool ContainsValue(U value) => (internalDictionary.ContainsValue(value));

public override bool Equals(object obj) => (internalDictionary.Equals(obj));

public IEnumerator GetEnumerator() => (internalDictionary.GetEnumerator());

public override int GetHashCode() => (internalDictionary.GetHashCode());

public void GetObjectData(SerializationInfo info, StreamingContext context)
{
    internalDictionary.GetObjectData(info, context);
}

public void OnDeserialization(object sender)
{
    internalDictionary.OnDeserialization(sender);
}

public override string ToString() => (internalDictionary.ToString());

public bool TryGetValue(T key, out U value) =>
      (internalDictionary.TryGetValue(key, out value));

public void Remove(T key)
{
    internalDictionary.Remove(key);
```

```
        }

        public void Clear()
        {
            internalDictionary.Clear();
        }
    }
```

2.4.3　讨论

MinMaxValueDictionary 类包装了 Dictionary<T,U> 类，因此它可以限制允许值的范围。下面的代码定义了 MinMaxValueDictionary 类的重载构造函数。

```
public MinMaxValueDictionary(U minValue, U maxValue)
```

该构造函数允许设置值的范围。它的参数包括以下两个。

- minValue

  可以作为值添加到键值对中类型 U 的最小值。

- maxValue

  可以作为值添加到键值对中类型 U 的最大值。

这些值可通过 MinMaxValueDictionary<T,U> 类上的 MinValue 和 MaxValue 属性获得。

重写的索引器拥有 get 和 set 访问器。get 返回与提供的 key 匹配的值。set 检查 value 参数，以确定它是否在之前设置的 minValue 和 maxValue 字段的边界内。

Add 方法为其 value 参数接受一个类型 U，并执行与索引器上的 set 访问器相同的测试。如果测试通过，就把整数添加到 MinMaxValueDictionary 中。

2.4.4　参考

MSDN 文档中的“Dictionary<T,U> 类”主题。

2.5　在应用程序会话间持久化一个集合

2.5.1　问题

你有一个诸如 ArrayList、List<T>、Hashtable 或 Dictionary<T,U> 这样的集合，在其中存储应用程序信息。可以使用该信息将应用程序的环境定制成最后已知的设置（例如，窗口大小、窗口位置和当前显示的工具栏）。也可以用它来允许用户在上一次关闭应用程序的相同位置启动应用程序。换句话说，如果用户正在编辑发票并且需要在晚上关闭计算机，当下次启动应用程序时，它将准确地知道初始要显示哪张发票。

2.5.2　解决方案

在对象与文件之间进行序列化 / 反序列化。

```
public static void SerializeToFile<T>(T obj, string dataFile)
{
    using (FileStream fileStream = File.Create(dataFile))
    {
        BinaryFormatter binSerializer = new BinaryFormatter();
        binSerializer.Serialize(fileStream, obj);
    }
}

public static T DeserializeFromFile<T>(string dataFile)
{
    T obj = default(T);
    using (FileStream fileStream = File.OpenRead(dataFile))
    {
        BinaryFormatter binSerializer = new BinaryFormatter();
        obj = (T)binSerializer.Deserialize(fileStream);
    }
    return obj;
}
```

2.5.3 讨论

dataFile 参数接受一个字符串值作为文件名。SerializeToFile<T> 方法接受一个对象并尝试将其序列化到一个文件。相反，DeserializeFromFile<T> 方法从 SaveObj<T> 方法创建的文件中获取序列化的对象。

例 2-3 展示了如何使用这些方法序列化一个 ArrayList 对象（注意，这适用于标记有 SerializableAttribute 的任何类型）。

例 2-3：在应用程序会话间持久化一个集合

```
ArrayList HT = new ArrayList() {"Zero","One","Two"};

foreach (object O in HT)
    Console.WriteLine(O.ToString());
SerializeToFile<ArrayList>(HT, "HT.data");

ArrayList HTNew = new ArrayList();
HTNew = DeserializeFromFile<ArrayList>("HT.data");
foreach (object O in HTNew)
    Console.WriteLine(O.ToString());
```

如果在应用程序中的特定时刻把对象序列化到磁盘，以后就可以反序列化它们并恢复到一种已知的状态，例如意外关机时。

你也可以在对象与字节流间进行序列化 / 反序列化以存储到独立存储或远程存储。

```
public static byte[] Serialize<T>(T obj)
{
    using (MemoryStream memStream = new MemoryStream())
    {
        BinaryFormatter binSerializer = new BinaryFormatter();
        binSerializer.Serialize(memStream, obj);
        return memStream.ToArray();
```

```
    }
}

public static T Deserialize<T>(byte[] serializedObj)
{
    T obj = default(T);
    using (MemoryStream memStream = new MemoryStream(serializedObj))
    {
        BinaryFormatter binSerializer = new BinaryFormatter();
        obj = (T)binSerializer.Deserialize(memStream);
    }
    return obj;
}
```

> 如果你依靠序列化的对象来存储持久信息，就需要清楚在部署应用程序的新版本时将要做什么。应该预先计划一种策略，确保序列化的类型不会发生变化；或者预先计划一种技术，用于处理所发生的变化。否则，在部署更新版本时将会遇到严重的问题。查阅 MSDN 中的文章“版本容错序列化”以了解处理这一情况的理念和最佳实践。

2.5.4 参考

MSDN 文档中的“ArrayList 类”“Hashtable 类”“List<T> 类”“Dictionary<T,U> 类”“File 类”“版本容错序列化”和“BinaryFormatter 类”主题。

2.6 测试Array或List<T>中的每个元素

2.6.1 问题

你需要采用一种容易的方法来测试 Array 或 List<T> 中的每个元素。这种测试的结果应该指示集合中的所有元素都通过了测试，或者集合中至少有一个元素没有通过测试。

2.6.2 解决方案

使用 TrueForAll 方法，代码如下所示。

```
// 创建一个字符串列表
List<string> strings = new List<string>() {"one",null,"three","four"};

// 确定列表中是否包含null值
string str = strings.TrueForAll(delegate(string val)
{
    if (val == null)
        return false;
    else
        return true;
}).ToString();
```

```
// 显示结果
Console.WriteLine(str);
```

2.6.3 讨论

Array 和 List<T> 类上添加的 TrueForAll 方法允许你轻松地为这些集合中的元素建立测试。2.6.2 节中的代码测试所有元素，以确定是否有任何元素为 null。你可以像这样轻松地建立测试，用以确定：

- 是否有任何数值元素大于指定的最大值；
- 是否有任何数值元素小于指定的最小值；
- 是否有任何字符串元素包含一组指定的字符；
- 是否有任何数据对象填充了它们的所有字段；
- 你可能提出的任何其他问题。

TrueForAll 方法接受一个名为 match 的泛型委托 Predicate<T>，并返回一个布尔值。

```
public bool TrueForAll(Predicate<T> match)
```

match 参数确定 TrueForAll 方法是否应该返回 true 或 false。

TrueForAll 方法实质上由一个循环组成，该循环用于遍历集合中的所有元素。在这个循环内调用 match 委托。如果该委托返回 true，就继续处理集合中的下一个元素。如果该委托返回 false，就停止处理，并且由 TrueForAll 方法返回 false。如果 TrueForAll 方法遍历完集合中的所有元素，并且 match 委托没有为任何元素返回一个 false 值，TrueForAll 方法就会返回 true。

并不存在 FalseForAll 方法，不过你可以逆转逻辑并且使用 TrueForAll 来完成同样的事情。

```
List<string> strings = new List<string>() {null, null, null, null};

// 确定列表中是否全部为null值
string str = strings.TrueForAll(delegate(string val)
{
    if (val == null)
        return true;
    else
        return false;
}).ToString();

// 显示结果
Console.WriteLine(str);
```

另外一个要考虑的因素是 TrueForAll 在第一次条件不为 true 时停止。这意味着并不是所有的节点都进行了检查。假若有一个包含文件或资源的数组需要进行处理或释放，你将需要迭代所有元素，即使在检查过程中执行的操作失败了也是如此。在这种情况下，你想注意是否其中的任何元素失败了，但访问 Array 或 List<T> 中的每个元素并执行操作仍然是很重要的。

2.6.4　参考

MSDN 文档中的“Array 类”“List<T> 类”和“TrueForAll 方法”主题。

2.7　创建自定义枚举器

2.7.1　问题

你需要向一个类中添加 foreach 支持，但是通常用于添加迭代器的方式（即在类型上实现 IEnumerable 并从一个成员函数返回指向这个 IEnumerable 的引用）并不足够灵活。除了简单地从第一个元素迭代到最后一个元素之外，还需要从最后一个元素迭代到第一个元素，并且需要能够在每次迭代时跨越或跳过预定义数量的元素。你想使所有这些类型的迭代器可供你的类使用。

2.7.2　解决方案

例 2-4 中所示的 Container<T> 类充当一个名为 internalList 的私有 List<T> 的容器。我们实现了 Container，因此可以在 foreach 循环中使用它来遍历私有 internalList。

例 2-4：创建自定义迭代器

```
public class Container<T> : IEnumerable<T>
{
    public Container() { }

    private List<T> _internalList = new List<T>();

    // 这个迭代器从头到尾迭代每一个元素
    public IEnumerator<T> GetEnumerator() => _internalList.GetEnumerator();

    // 这个迭代器从尾到头迭代每一个元素
    public IEnumerable<T> GetReverseOrderEnumerator()
    {
        foreach (T item in ((IEnumerable<T>)_internalList).Reverse())
            yield return item;
    }

    // 这个迭代器从头到尾迭代每一个元素,按预定义的大小步进
    public IEnumerable<T> GetForwardStepEnumerator(int step)
    {
        foreach (T item in _internalList.EveryNthItem(step))
            yield return item;
    }

    // 这个迭代器从尾到头迭代每一个元素,按预定义的大小步进
    public IEnumerable<T> GetReverseStepEnumerator(int step)
    {
        foreach (T item in (
            (IEnumerable<T>)_internalList).Reverse().EveryNthItem(step))
            yield return item;
    }
```

```
        #region IEnumerable Members

        IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

        #endregion

        public void Clear()
        {
            _internalList.Clear();
        }

        public void Add(T item)
        {
            _internalList.Add(item);
        }

        public void AddRange(ICollection<T> collection)
        {
            _internalList.AddRange(collection);
        }
    }
```

2.7.3 讨论

迭代器提供了一种容易的方法，使用我们熟悉的 foreach 循环构造在对象内从一个数据项移到另一个数据项。该对象可以是数组、集合或者某种其他类型的容器。这类似于使用 for 循环手动遍历数组中包含的每个数据项。事实上，可以建立一个使用 for 循环或任何其他循环构造的迭代器，将循环构造作为输出对象中的每个数据项的机制。事实上，甚至不必使用循环构造。下面的代码是完全有效的。

```
public static IEnumerable<int> GetValues()
{
    yield return 10;
    yield return 20;
    yield return 30;
    yield return 100;
}
```

利用 foreach 循环，不必担心要检查列表的末尾，因为它不能越出列表的边界。关于 foreach 循环和迭代器的最佳优点是，不必知道如何访问其容器内的元素列表。事实上，你甚至不必访问元素的列表，因为容器上实现的迭代器成员会为你做这项工作。

为了了解 foreach 在这里的作用，让我们来看一下遍历 Container 类的代码。

```
// 迭代容器对象
foreach (int i in container)
    Console.WriteLine(i);
```

在这段代码运行时，foreach 将执行以下几个操作。

(1) 使用 IEnumerator.GetEnumerator() 从容器中获取枚举器。

(2) 访问 IEnumerator.Current 属性以获取当前对象（int），并将其置于 i 中。
(3) 调用 IEnumerator.MoveNext()。如果 MoveNext 返回 true，就返回到第 2 步，否则结束循环。

Container 类包含一个数据项的私有 List，名为 internalList。这个类中有以下 4 个迭代器成员。

- GetEnumerator
- GetReverseOrderEnumerator
- GetForwardStepEnumerator
- GetReverseStepEnumerator

GetEnumerator 方法从第一个元素到最后一个元素遍历 internalList 中的每个元素。与其他迭代器类似，这个迭代器使用 for 循环输出 internalList 中的每个元素。

GetReverseOrderEnumerator 方法在其 get 访问器中实现了一个迭代器（set 访问器不能是迭代器）。这个迭代器在设计上非常类似于 GetEnumerator 方法，只不过 foreach 循环在相反方向上操作 internalList，这是使用 IEnumerable<T>.Reverse 扩展方法实现的。最后两个迭代器是 GetForwardStepEnumerator 和 GetReverseStepEnumeraror，在设计上分别类似于 GetEnumerator 和 GetReverseOrderEnumerator。其主要区别是，foreach 循环使用 EveryNthItem 扩展方法跳过 internalList 中指定数量的数据项。

```
public static IEnumerable<T> EveryNthItem<T>(this IEnumerable<T> enumerable,
    int step)
{
    int current = 0;
    foreach (T item in enumerable)
    {
        ++current;
        if (current % step == 0)
            yield return item;
    }
}
```

另外要注意，只有 GetEnumerator 方法必须返回 IEnumerator<T> 接口，其他三个迭代器都必须返回 IEnumerable<T> 接口。

为了从第一个元素到最后一个元素遍历 Container 对象中的每个元素，可使用以下代码。

```
Container<int> container = new Container<int>();
    //……将数据添加到容器……
foreach (int i in container)
    Console.WriteLine(i);
```

为了从最后一个元素到第一个元素遍历 Container 对象中的每个元素，可使用以下代码。

```
Container<int> container = new Container<int>();
    //……将数据添加到容器……
foreach (int i in container.GetReverseOrderEnumerator())
    Console.WriteLine(i);
```

为了从第一个元素到最后一个元素遍历 Container 对象中的每个元素，同时跳到每隔一个

元素的下一个元素，可使用以下代码。

```
Container<int> container = new Container<int>();
    //……将数据添加到容器……
foreach (int i in container.GetForwardStepEnumerator(2))
    Console.WriteLine(i);
```

为了从最后一个元素到第一个元素遍历 Container 对象中的每个元素，同时跳到所有每隔两个元素的第三个元素，可使用以下代码。

```
Container<int> container = new Container<int>();
    //……将数据添加到容器……
foreach (int i in container.GetReverseStepEnumerator(3))
    Console.WriteLine(i);
```

在后两个示例中，迭代器方法接受一个整数值 step，它确定将跳过多少个元素。

最后一点关于 yield 的说明，虽然在 lock 语句内使用 yield 在技术上是可行的（参见 9.9.3 节了解 lock 的更多信息），但是你应当避免这样做，因为它可能导致应用程序内的死锁。yield 语句后执行的代码可能会将 lock 带出并导致死锁。lock 内的代码可能会在另一个线程上恢复（因为在 yield 之后，代码并不需要在原线程上恢复），因此可能会在建立锁的线程之外的另一个不同的线程解锁。

2.7.4　参考

MSDN 文档中的“迭代器”“yield”“IEnumerator 接口”“IEnumerable(T) 接口”和“IEnumerable 接口”主题，以及范例 9.9（即 9.9 节）。

2.8　处理finally语句块和迭代器

2.8.1　问题

你向迭代器中添加了一个 try-finally 语句块，并且注意到 finally 块并没有按所想的那样执行。

2.8.2　解决方案

在 GetEnumerator 迭代器中用一个 try 块包围迭代代码，并在该 try 块后接一个 finally 块，代码如下所示。

```
public class StringSet : IEnumerable<string>
{
    private List<string> _items = new List<string>();

    public void Add(string value)
    {
        _items.Add(value);
    }
```

```
    public IEnumerator<string> GetEnumerator()
    {
        try
        {
            for (int index = 0; index < _items.Count; index++)
            {
                yield return (_items[index]);
            }
        }
        // 不能在迭代器中使用catch语句块
        finally
        {
            // 仅在foreach循环结束后执行(包括yield中断时)
            Console.WriteLine("In iterator finally block");
        }
    }

    #region IEnumerable Members

    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

    #endregion
}
```

调用这个迭代器的 foreach 代码如下所示。

```
//创建一个StringSet对象并填入数据
StringSet strSet =
    new StringSet()
        {"item1",
            "item2",
            "item3",
            "item4",
            "item5"};

// 使用GetEnumerator迭代器
foreach (string s in strSet)
    Console.WriteLine(s);
```

运行这段代码时，将显示以下输出。

```
item1
item2
item3
item4
item5
In iterator finally block
```

2.8.3 讨论

你可能认为这段代码的输出将在显示 strSet 对象中的每个数据项之后显示 In iterator finally block 字符串。不过，这不是在迭代器中处理 finally 块的方式。仅当迭代完成之后，代码执行离开 foreach 循环时（比如，当遇到 break、return 或 throw 语句时），或者当执行 yield break 语句时，才会调用与迭代器成员体内包含 yield 返回语句的 try 块关

联的所有 finally 块，从而有效地终止迭代器。

为了查看迭代器如何处理 catch 和 finally 语句块（注意，包含 yield 语句的 try 块内不能包含 catch 块），思考以下代码。

```
///创建一个StringSet对象并填入数据
StringSet strSet =
    new StringSet()
        {"item1",
            "item2",
            "item3",
            "item4",
            "item5"};

// 显示StringSet对象中的所有数据
try
{
    foreach (string s in strSet)
    {
        try
        {
            Console.WriteLine(s);
            //强制引发异常
            //throw new Exception();
        }
        catch (Exception)
        {
            Console.WriteLine("In foreach catch block");
        }
        finally
        {
            // 每次迭代时执行
            Console.WriteLine("In foreach finally block");
        }
    }
}
catch (Exception)
{
    Console.WriteLine("In outer catch block");
}
finally
{
    // 每次迭代时执行
    Console.WriteLine("In outer finally block");
}
```

假定使用了原始的 StringSet.GetEnumerator 方法（即包含 try-finally 语句块的方法），将会看到以下行为。

如果没有异常发生，将看到如下输出。

```
item1
In foreach finally block
item2
In foreach finally block
```

```
item3
In foreach finally block
item4
In foreach finally block
item5
In foreach finally block
In iterator finally block
In outer finally block
```

我们看到，每次迭代都会执行 foreach 循环内的 finally 语句块。不过，仅当所有迭代都完成之后，才会执行迭代器内的 finally 语句块。另外，注意到迭代器的 finally 语句块将会在包装 foreach 循环的 finally 语句块之前执行。

如果在处理第二个元素期间，迭代器自身发生异常，那么将会显示如下输出。

```
item1
In foreach finally block
     (Exception occurs here...)
In iterator finally block
In outer catch block
In outer finally block
```

我们注意到，一旦引发异常，就会执行迭代器内的 finally 语句块。如果你只需在异常发生之后执行清理工作，这会是很有用的。如果没有异常发生，那么直到迭代器执行完成才会执行 finally 语句块。在迭代器的 finally 语句块执行之后，foreach 循环外部的 catch 语句块将会捕获异常。此时可以处理或重新引发异常。一旦处理完这个 catch 语句块，就会执行外层 finally 语句块。

注意，永远不给 foreach 循环内的 catch 语句块提供处理异常的机会。这是由于对应的 try 语句块没有包含对迭代器的调用。

如果在 foreach 循环内处理第二个元素期间发生异常，那么将会显示如下输出。

```
item1
In foreach finally block
     (Exception occurs here...)
In foreach catch block
In foreach finally block
In iterator finally block
In outer finally block
```

注意，在这种情况下，首先执行 foreach 循环内的 catch 和 finally 语句块，然后执行迭代器的 finally 语句块，最后执行外层 finally 语句块。

理解迭代器内的 catch 和 finally 语句块的工作方式将帮助你在正确的位置添加 catch 和 finally 语句块。如果需要在迭代完成之后立即执行一次 finally 语句块，可以将该 finally 语句块添加到迭代器方法中。不过，如果希望每次迭代都执行 finally 语句块，就需要将 finally 语句块置于 foreach 循环体内。

如果需要在迭代器异常发生之后立即捕获它们，就应该考虑将 foreach 循环包装在一个 try-catch 语句块中。foreach 循环内的任何 try-catch 语句块都将错过从迭代器引发的异常。

2.8.4 参考

MSDN 文档中的“try-catch”“迭代器”“yield”“IEnumerator 接口”和“IEnumerable 接口”主题。

2.9 在类中实现嵌套的foreach功能

2.9.1 问题

你需要一个类，它包含一个对象列表，其中每个对象也包含一个对象列表。你希望以如下方式使用嵌套的 foreach 循环遍历外层和内层列表中的所有对象。

```
foreach (Group<Item> subGroup in topLevelGroup)
{
    // 操作组
    foreach (Item item in subGroup)
    {
        // 操作数据项
    }
}
```

2.9.2 解决方案

在类上实现 IEnumerable<T> 接口。例 2-5 中所示的 Group 类包含一个可以保存 Group 对象的 List<T>，并且每个 Group 对象都包含一个 List<Item>。

例 2-5：在类中实现 foreach 功能

```
public class Group<T> : IEnumerable<T>
{
    public Group(string name)
    {
        this.Name = name;
    }

    private List<T> _groupList = new List<T>();

    public string Name { get; set; }

    public int Count => _groupList.Count;

    public void Add(T group)
    {
        _groupList.Add(group);
    }

    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

    public IEnumerator<T> GetEnumerator() => _groupList.GetEnumerator();
}
```

```
public class Item
{
    public Item(string name, int location)
    {
        this.Name = name;
        this.Location = location;
    }
    public string Name { get; set; }
    public int Location { get; set; }
}
```

2.9.3 讨论

在 C# 语言中，使用迭代器在类中构建功能从而用 foreach 循环遍历它变得容易多了。在 .NET Framework 3.0 之前的版本中，不仅必须要在希望可枚举的类型上实现 IEnumerable 接口，而且要在嵌套类上实现 IEnumerator 接口。然后必须在这个嵌套类中手动编写 MoveNext 和 Reset 方法以及 Current 属性。迭代器允许你把编写这个嵌套类的工作移交给 C# 编译器。如果编写一个旧式的枚举器，代码看起来将如下所示。

```
public class GroupEnumerator<T> : IEnumerator
{
    public T[] _items;

    int position = -1;

    public GroupEnumerator(T[] list)
    {
        _items = list;
    }

    public bool MoveNext()
    {
        position++;
        return (position < _items.Length);
    }

    public void Reset()
    {
        position = -1;
    }

    public object Current
    {
        get
        {
            try
            {
                return _items[position];
            }
            catch (IndexOutOfRangeException)
            {
                throw new InvalidOperationException();
            }
```

```
        }
    }
}
```

这会在 Group<T> 类上修改 IEnumerator.GetEnumerator 方法，如下所示。

```
IEnumerator IEnumerable.GetEnumerator() =>
    new GroupEnumerator<T>(_groupList.ToArray());
```

利用该方法的代码如下所示。

```
IEnumerator enumerator = ((IEnumerable)hierarchy).GetEnumerator();
while (enumerator.MoveNext())
{
    Console.WriteLine(((Group<Item>)enumerator.Current).Name);
    foreach (Item i in ((Group<Item>)enumerator.Current))
    {
        Console.WriteLine(i.Name);
    }
}
```

如果你不必这样做，难道不感到高兴吗？把它留给编译器，编译器十分擅长为你编写这些代码。

为了使 foreach 循环能够使用一个类，需要包含一个迭代器。迭代器可以是一个方法、一个运算符重载或者一个属性的 get 访问器。它返回一个 System.Collections.IEnumerator、System.Collections.Generic.IEnumerator<T>、System.Collections.IEnumerable 或 System.Collections.Generic.IEnumerable<T>，或者包含至少一个 yield 语句。

用于本范例的代码划分在两个类中。容器类是 Group 类，包含 Group<Item> 对象的 List。Group 对象也包含一个 List，但是这个 List 包含 Item 对象。为了枚举包含的列表，Group 类实现了 IEnumerable 接口。因此它包含一个 GetEnumerator 迭代器方法，该方法返回一个 IEnumerator。类结构如下所示。

```
Group (Implements IEnumerable<T>)
  Group (Implements IEnumerable<T>)
     Item
```

通过检查 Group 类，你可以看出如何构造可供 foreach 循环使用的类。这个类包含以下各项。

- 一个简单的 List<T>，将由类的枚举器遍历它。
- 一个属性 Count，它将返回 List<T> 中的元素个数。
- 一个迭代器方法 GetEnumerator，它是由 IEnumerable<T> 接口定义的。该方法将在 foreach 循环执行每次迭代时输出一个特定的值。
- 一个方法 Add，它会将类似 Subgroup 这样的实例添加到 List<T> 中。
- 一个方法 GetGroup，它将从 List<T> 中返回一个类型化的实例，如 Subgroup。

为了创建 Subgroup 类，可以遵循与 Group 类相同的模式，只不过 Subgroup 类包含一个 List<Item>。

最后一个类是 Item。该类位于这个结构的最低级别上，并且包含了数据。它被组织在 Subgroup 对象内，所有这些 Subgroup 对象都包含在 Group 对象中。这个类并没有任何与众不同的地方，它只是包含数据以及用于设置和获取该数据的方式。

使用这些类十分简单。下面的方法显示了如何创建包含多个 Subgroup 对象的 Group 对象，这些 Subgroup 对象依次又包含多个 Item 对象。

```
public static void CreateNestedObjects()
{
    Group<Group<Item>> hierarchy =
        new Group<Group<Item>>("root") {
            new Group<Item>("subgroup1"){
                new Item("item1",100),
                new Item("item2",200)},
            new Group<Item>("subgroup2"){
                new Item("item3",300),
                new Item("item4",400)}};

    IEnumerator enumerator = ((IEnumerable)hierarchy).GetEnumerator();
    while (enumerator.MoveNext())
    {
        Console.WriteLine(((Group<Item>)enumerator.Current).Name);
        foreach (Item i in ((Group<Item>)enumerator.Current))
        {
            Console.WriteLine(i.Name);
        }
    }

    // 读回数据
    DisplayNestedObjects(hierarchy);
}
```

CreateNestedObjects 方法首先创建 Group 类的一个 hierarchy 对象，然后在其中创建两个子组，名为 subgroup1 和 subgroup2。每个子组对象又被依次填充了两个 Item 对象，分别名为 iteml、item2、item3 和 item4。

下一个方法显示了如何读取在 CreateNestedObjects 方法中创建的 Group 对象内包含的所有 Item 对象。

```
private static void DisplayNestedObjects(Group<Group<Item>> topLevelGroup)
{
    Console.WriteLine($"topLevelGroup.Count: {topLevelGroup.Count}");
    Console.WriteLine($"topLevelGroupName:  {topLevelGroup.Name}");

    // 外部的foreach迭代topLevelGroup对象中的所有对象
    foreach (Group<Item> subGroup in topLevelGroup)
    {
        Console.WriteLine($"\tsubGroup.SubGroupName:  {subGroup.Name}");
        Console.WriteLine($"\tsubGroup.Count: {subGroup.Count}");

        // 内部的foreach迭代当前SubGroup对象中的所有对象
        foreach (Item item in subGroup)
        {
            Console.WriteLine($"\t\titem.Name:     {item.Name}");
```

```
                Console.WriteLine($"\t\titem.Location: {item.Location}");
            }
        }
    }
```

该方法将显示如下结果。

```
topLevelGroup.Count: 2
topLevelGroupName:  root
        subGroup.SubGroupName:  subgroup1
        subGroup.Count: 2
                item.Name:     item1
                item.Location: 100
                item.Name:     item2
                item.Location: 200
        subGroup.SubGroupName:  subgroup2
        subGroup.Count: 2
                item.Name:     item3
                item.Location: 300
                item.Name:     item4
                item.Location: 400
```

在这里可以看到，外层 foreach 循环用于遍历顶级 Group 对象中存储的所有 Subgroup 对象，内层 foreach 循环用于遍历当前 Subgroup 对象中存储的所有 Item 对象。

2.9.4 参考

MSDN 文档中的“迭代器”“yield”“IEnumerator 接口”“IEnumerable(T) 接口”和“IEnumerable 接口”主题。

2.10 使用线程安全的字典进行并发访问，不手动加锁

2.10.1 问题

你需要创建一个从多个线程中并发读写的键值对的集合，并且不需要手动使用同步原语来保护它。

2.10.2 解决方案

使用 ConcurrentDictionary<Tkey, TValue> 容纳数据项并以线程安全的方式来访问它们。

举例来说，考虑一个模拟情形，球迷到一个体育馆去观看他们喜爱的体育赛事（超级碗、世界杯、世界系列赛或者 ICC 世界板球联盟锦标赛），并且场馆仅有若干个入口。

首先，我们需要一些球迷并且记录他们的姓名、何时入场以及通过哪个入口入场。

```
public class Fan
{
```

```
    public string Name { get; set; }
    public DateTime Admitted { get; set; }
    public int AdmittanceGateNumber { get; set; }
}

// 设置一个参加活动的球迷名单
List<Fan> fansAttending = new List<Fan>();
for (int i = 0; i < 100; i++)
    fansAttending.Add(new Fan() { Name = "Fan" + i });
Fan[] fans = fansAttending.ToArray();
```

每个入口每次只能有一个人入场，并且在如此拥挤的活动中，入口通常都有摄像头监控。我们将用一个静态的 ConcurrentDictionary<int, Fan> 来表示体育馆的入口以及当前在入口的 Fan，并用一个静态布尔变量（monitorGates）来表明何时入口不再需要监控（也就是所有球迷都已入场的时候）。

```
private static ConcurrentDictionary<int, Fan> stadiumGates =
    new ConcurrentDictionary<int, Fan>();

private static bool monitorGates = true;
```

假设活动现场共有 10 个入口（gateCount），为每个入口使用 AdmitFans 方法启动一个 Task，让每个入口能够进入若干球迷。还要为每个入口使用 MonitorGate 方法启动一个对应的 Task，将安全监控打开。当所有球迷都入场后，停止监控入口。

```
int gateCount = 10;
Task[] entryGates = new Task[gateCount];
Task[] securityMonitors = new Task[gateCount];

for (int gateNumber = 0; gateNumber < gateCount; gateNumber++)
{
    //有趣的事实：
    //你也许会认为当for循环中的gateNumber变化时，
    //允许Fan进入的Task在创建时能够捕获
    //Task在创建时那一刻的值(0、1、2等)
    //这意味着就算你为入口0启动了一个Task，
    //随着Task创建后循环的继续，
    //它将得到一个值为9的gateNumber变量，
    //为了处理这一情况,我们将值赋予一个局部变量以修正作用域，
    //并且你想要的值可以被Task正确地捕获
    int GateNum = gateNumber;
    int GateCount = gateCount;
    Action action = delegate () { AdmitFans(fans, GateNum, GateCount); };
    entryGates[gateNumber] = Task.Run(action);
}

for (int gateNumber = 0; gateNumber < gateCount; gateNumber++)
{
    int GateNum = gateNumber;
    Action action = delegate () { MonitorGate(GateNum); };
    securityMonitors[gateNumber] = Task.Run(action);
}
```

```
    await Task.WhenAll(entryGates);

    // 关闭监控
    monitorGates = false;
```

AdmitFans 执行准许一部分球迷通过的工作，它使用 ConcurrentDictionary<TKey, TValue>.AddOrUpdate 方法表示 Fan 位于入口处，使用 ConcurrentDictionary<TKey, TValue>.TryRemove 方法表示该 Fan 被准许进入活动现场，代码如下所示。

```
private static void AdmitFans(Fan[] fans, int gateNumber, int gateCount)
{
    Random rnd = new Random();
    int fansPerGate = fans.Length / gateCount;
    int start = gateNumber * fansPerGate;
    int end = start + fansPerGate - 1;
    for (int f = start; f <= end; f++)
    {
        Console.WriteLine($"Admitting {fans[f].Name} through gate {gateNumber}");
        var fanAtGate =
            stadiumGates.AddOrUpdate(gateNumber, fans[f],
                (key, fanInGate) =>
                {
                    Console.WriteLine($"{fanInGate.Name} was replaced by " +
                        $"{fans[f].Name} in gate {gateNumber}");
                    return fans[f];
                });
        // 执行搜身检查并检查门票
        Thread.Sleep(rnd.Next(500, 2000));
        // 让他们通过入口
        fans[f].Admitted = DateTime.Now;
        fans[f].AdmittanceGateNumber = gateNumber;
        Fan fanAdmitted;
        if(stadiumGates.TryRemove(gateNumber, out fanAdmitted))
            Console.WriteLine($"{fanAdmitted.Name} entering event from gate " +
                $"{fanAdmitted.AdmittanceGateNumber} on " +
                $"{fanAdmitted.Admitted.ToShortTimeString()}");
        else // 如果不能允许他们进入,保安必须扣留他们
            Console.WriteLine($"{fanAdmitted.Name} held by security " +
                $"at gate {fanAdmitted.AdmittanceGateNumber}");
    }
}
```

MonitorGate 通过使用 TryGetValue 方法检查 ConcurrentDictionary<TKey, TValue> 来观察指定的入口（gateNumber）。MonitorGate 将持续监控直到 monitorGates 标志设置为 false（在 Admit Fan Task 完成之后）。

```
private static void MonitorGate(int gateNumber)
{
    Random rnd = new Random();
    while (monitorGates)
    {
        Fan currentFanInGate;
        if (stadiumGates.TryGetValue(gateNumber, out currentFanInGate))
            Console.WriteLine($"Monitor: {currentFanInGate.Name} is in Gate " +
```

```
                $"{gateNumber}");
        else
            Console.WriteLine($"Monitor: No fan is in Gate {gateNumber}");

        //等待然后再次检查入口
        Thread.Sleep(rnd.Next(500, 5000));
    }
}
```

2.10.3 讨论

`ConcurrentDictionary<TKey, TValue>` 位于 `System.Collections.Concurrent` 命名空间内，在 .NET 4.0 及以上版本中可用。它是一种在多重读写的情形下最有用的集合。如果你在初始化后仅需要多重读，那么 `ImmutableDictionary<TKey, TValue>` 是一个更好的选择。因为集合在初始化后是不可变的，所以它是一种速度更快的可读集合。当你操作一个不可变字典时，会创建原始字典的一个新副本。这一过程的代价很高，因此应该仅将不可变集合用于加载一次后反复读写的情况。

注意这些 `Immutable` 类并不是核心 .NET Framework 类库中的一部分；它们位于 NuGet 包 `Microsoft.Bcl.Immutable` 的 `System.Collections.Immutable` 程序集内，你需要在你的应用程序中包含这个包。

`ConcurrentDictionary<TKey, TValue>` 包含一些会导致整个集合加锁的属性和方法，因为它们需要同时操作集合中的所有数据项。以下这些属性会返回数据的“即时”快照。

- `Count` 属性
- `Keys` 属性
- `Values` 属性
- `ToArray` 方法

数据的一个“即时”表现意味着在枚举时并不一定获得当前数据。如果需要字典中绝对的当前数据，请使用 `GetEnumerator` 方法，它提供了一个用于遍历字典中的键值对的枚举器。因为 `GetEnumerator` 保证了枚举器在并发更新时都可以安全使用，并且不需要加锁，你能够在底层数据改变时访问它。参考 MSDN 文档中的“`IEnumerable<T>.GetEnumerator`”主题可获得更详细的介绍。

LINQ 经常使用 `GetEnumerator` 以达成它的目标，因此你可以用一个 LINQ 的 `Select` 方法来使用 `Keys` 和 `Values` 属性，以避免加锁带来的不利影响。

```
var keys = stadiumGates.Select(gate => gate.Key);
var values = stadiumGates.Select(gate => gate.Value);
```

当用于计数时，LINQ 被优化为在支持 `ICollection` 接口的集合上使用 `Count` 属性，但在本例中这不是我们想要的。下面的例子展示了 `Count` 方法的逻辑；如果可枚举参数支持 `ICollection` 或者 `ICollection<T>`，就使用 `Count` 属性。

```
public static int Count<TSource>(this IEnumerable<TSource> source)
{
    if (source == null)
```

```
        {
            throw Error.ArgumentNull("source");
        }
        ICollection<TSource> tSources = source as ICollection<TSource>;
        if (tSources != null)
        {
            return tSources.Count;
        }
        ICollection collections = source as ICollection;
        if (collections != null)
        {
            return collections.Count;
        }
        int num = 0;
        using (IEnumerator<TSource> enumerator = source.GetEnumerator())
        {
            while (enumerator.MoveNext())
            {
                num++;
            }
        }
        return num;
    }
```

要绕过这个问题，可获取一个不支持 ICollection 和 ICollection<T> 的可枚举数据项集合，然后对其调用 Count()。

```
var count = stadiumGates.Select(gate => gate).Count();
```

Select 调用返回一个 System.Linq.Enumerable.WhereSelectEnumerableIterator，它不支持 ICollection 或 ICollection<T>，但是支持 GetEnumerator。

表 2-1 总结了 ConcurrentDictionary<TKey, TValue> 的主要数据操作方法概况。

表2-1：ConcurrentDictionary的操作方法

方　法	何时使用
TryAdd	仅当键不存在时添加一个新数据项
TryUpdate	如果当前值可用，将现有键更新为新值
Indexing	无条件地设置字典中的一个键 / 值，无论键存在与否
AddOrUpdate	用一个委托设置字典中的一个键 / 值，可以根据键添加还是更新设置不同记录
GetOrAdd	获得一个键的值，或者初始化键的值并返回该值（延迟初始化）
TryGetValue	获得键的值，或者返回 false
TryRemove	移除键对应的值，或者返回 false

示例代码开始的输出看起来如下所示。

```
Admitting Fan0 through gate 0
Admitting Fan10 through gate 1
Admitting Fan20 through gate 2
Admitting Fan30 through gate 3
Admitting Fan40 through gate 4
```

```
Fan0 entering event from gate 0 on 6:00 PM
Admitting Fan1 through gate 0
Fan20 entering event from gate 2 on 6:00 PM
Fan10 entering event from gate 1 on 6:00 PM
Admitting Fan11 through gate 1
Fan30 entering event from gate 3 on 6:00 PM
Admitting Fan31 through gate 3
Admitting Fan21 through gate 2
Fan40 entering event from gate 4 on 6:00 PM
Admitting Fan41 through gate 4
Admitting Fan50 through gate 5
Fan11 entering event from gate 1 on 6:00 PM
Admitting Fan12 through gate 1
Fan1 entering event from gate 0 on 6:00 PM
```

示例代码运行中途监控的输出看起来如下所示。

```
...
Admitting Fan17 through gate 1
Fan6 entering event from gate 0 on 6:00 PM
Admitting Fan7 through gate 0
Fan26 entering event from gate 2 on 6:00 PM
Admitting Fan27 through gate 2
Fan36 entering event from gate 3 on 6:00 PM
Admitting Fan37 through gate 3
Monitor: Fan17 is in Gate 1
Fan83 entering event from gate 8 on 6:00 PM
Admitting Fan55 through gate 5
Fan17 entering event from gate 1 on 6:00 PM
Admitting Fan18 through gate 1
...
```

示例代码最后的输出看起来如下所示。

```
...
Monitor: Fan97 is in Gate 9
Monitor: No fan is in Gate 0
Fan77 entering event from gate 7 on 6:00 PM
Admitting Fan78 through gate 7
Monitor: No fan is in Gate 1
Fan97 entering event from gate 9 on 6:00 PM
Admitting Fan98 through gate 9
Monitor: No fan is in Gate 2
Monitor: No fan is in Gate 3
Monitor: No fan is in Gate 4
Monitor: No fan is in Gate 5
Monitor: No fan is in Gate 6
Monitor: Fan78 is in Gate 7
Monitor: No fan is in Gate 8
Fan78 entering event from gate 7 on 6:00 PM
Admitting Fan79 through gate 7
Monitor: Fan98 is in Gate 9
Monitor: No fan is in Gate 2
Fan98 entering event from gate 9 on 6:00 PM
Admitting Fan99 through gate 9
```

```
Monitor: No fan is in Gate 1
Monitor: No fan is in Gate 2
Fan79 entering event from gate 7 on 6:00 PM
Fan99 entering event from gate 9 on 6:00 PM
```

2.10.4　参考

MSDN 文档中的“IEnumerable<T>.GetEnumerator”和“ConcurrentDictionary<TKey, TValue>”主题。

第 3 章

数据类型

3.0 简介

作为一种值类型，简单类型（simple type）是 C# 中内建类型的一个子集，不过事实上这些类型被定义为 .NET Framework 类库（.NET FCL）的一部分。简单类型由若干数字类型和一个 bool 类型构成。这些数字类型包括一个十进制类型（decimal）、九个整数类型（byte、char、int、long、sbyte、short、unit、ulong 和 ushort）以及两个浮点类型（float 和 double）。表 3-1 列出了 .NET Framework 中的简单类型及其完全限定名。

表3-1：简单数据类型

全　　名	别　　名	取值范围
System.Boolean	bool	true 或 false
System.Byte	byte	0~255
System.SByte	sbyte	-128~127
System.Char	char	0~65535
System.Decimal	decimal	-79,228,162,514,264,337,593,543,950,335~79,228,162,514,264,337,593,543,950,335
System.Double	double	-1.79769313486232e308~1.79769313486232e308
System.Single	float	-3.40282347E+38~3.40282347E+38
System.Int16	short	-32768~32767
System.Uint16	ushort	0~65535
System.Int32	int	-2,147,483,648~2,147,483,647
System.UInt32	uint	0~4,294,967,295

（续）

全　　名	别　　名	取值范围
System.Int64	long	-9,223,372,036,854,775,808~9,223,372,036,854,775,807
System.UInt64	ulong	0~18,446,744,073,709,551,615

当处理浮点数据类型时，精度可能比数值的范围更重要。浮点数据类型的精度如表 3-2 所示。

表3-2：浮点精度

浮点类型	精　　度
System.Single(float)	7 位
System.Double(double)	15~16 位
System.Decimal(decimal)	28~29 位

在使用浮点与使用十进制数之间进行决策时，要考虑以下两个方面。

- 浮点供科学家使用，设计用于表示物理学中精度和量值的整个范围上的不精确量。
- 十进制数供普通人使用，设计用于进行十进制计算，仅需要小数点之后少量的数字，或者用于需要准确记录每一分钱的情况（例如核对支票本）。

C# 为各种数据类型保留的关键字是针对类型的完全限定名称的简单别名。因此，用户使用类型名还是保留字是无关紧要的，C# 编译器将生成同样的代码。

要注意下列类型与*公共语言规范*（common language specification，CLS）并不兼容：sbyte、ubyte、uint 和 ulong。结果可能导致它们不被其他 .NET 语言支持。枚举隐式继承自 System.Enum，后者又继承自 System.ValueType。枚举有着单一的应用，描述一个特定组的数据项。例如，颜色 Red、Blue 和 Yellow 可由枚举 ShapeColor 定义；同样，形状 Square、Circle 和 Triangle 可以由枚举 Shape 定义。这些枚举如下所示。

```
enum ShapeColor
{
    Red, Blue, Yellow
}

enum Shape
{
    Square = 2, Circle = 4, Triangle = 6
}
```

枚举中的每个数据项接受一个数字值，无论你是否给它赋过值。由于编译器为枚举中的每个项自动添加以 0 开头并且递增 1 的数字，因此上述定义的 ShapeColor 枚举如果以下列方式定义，那么它将完全相同。

```
enum ShapeColor
{
    Red = 0,Blue = 1,Yellow = 2
}
```

枚举是一种很好的代码文档化工具。例如，如下编写的代码会更直观。

```
ShapeColor currentColor = ShapeColor.Red;
```

下面这样就不太直观。

```
int currentColor = 0;
```

这两种机制都能工作，但第一种方法容易阅读和理解，特别是对于一个接管其他人代码的新开发人员而言。枚举对保证 C# 中的类型安全也有益处，而使用原始的 int 不能提供数据安全。CLR 将枚举视为其基础类型的成员，因此对于所有语言它并非都是类型安全的。

3.1 把二进制数据编码为base64格式

3.1.1 问题

你有一个 byte[] 用于表示一些二进制信息，比如位图。你需要把该数据编码为一个字符串，以便可以通过不适合传输二进制的方式（比如电子邮件）发送它。

3.1.2 解决方案

使用 Convert 类的静态方法 Convert.ToBase64String，可以把 byte[] 编码为其对应的 String。

```
static class DataTypeExtMethods
{
    public static string Base64EncodeBytes(this byte[] inputBytes) =>
        (Convert.ToBase64String(inputBytes));
}
```

3.1.3 讨论

把字符串转换为其 base64 表示具有多种用途。它允许把二进制数据嵌入到非二进制文件中，比如 XML、电子邮件消息等。还可以用比十六进制编码更简洁的格式通过 HTTP、GET 和 POST 请求传输 base64 编码的数据。把数据转换成为 base64 格式只是使其混淆而不会对其加密，理解这一点是很重要的。为了安全地把数据从一个地方移到另一个地方，应该使用 FCL 中提供的加密算法。有关使用 FCL 加密类的示例，参见范例 11.4（即 11.4 节）。

Convert 类使得 byte[] 与 String 之间的编码成为一件简单的事情。这个方法的参数相当灵活，允许在输入字节数组中的任意位置开始和停止转换。

为了把位图文件编码为可以发送到某个目的地的字符串，可以使用 EncodeBitmapToString 方法。

```
public static string EncodeBitmapToString(string bitmapFilePath)
{
    byte[] image = null;
    FileStream fstrm =
        new FileStream(bitmapFilePath,
```

```
                    FileMode.Open, FileAccess.Read);
    using (BinaryReader reader = new BinaryReader(fstrm))
    {
        image = new byte[reader.BaseStream.Length];
        for (int i = 0; i < reader.BaseStream.Length; i++)
            image[i] = reader.ReadByte();
    }
    return image.Base64EncodeBytes();
}
```

> MIME 标准要求 base64 编码字符串每一行的长度为 76 个字符。为了在电子邮件消息中作为嵌入式 MIME 附件发送 `bmpAsString` 字符串，必须在每 76 个字符的边界上插入一个 CRLF。

将 base64 编码的字符串转换为适用于 MIME 的字符串的代码展示在下面的 `MakeBase64EncodedStringForMime` 方法中。

```
public static string MakeBase64EncodedStringForMime(string base64Encoded)
{
    StringBuilder originalStr = new StringBuilder(base64Encoded);
    StringBuilder newStr = new StringBuilder();
    const int mimeBoundary = 76;
    int cntr = 1;
    while ((cntr * mimeBoundary) < (originalStr.Length - 1))
    {
        newStr.AppendLine(originalStr.ToString(((cntr - 1) * mimeBoundary),
            mimeBoundary));
        cntr++;
    }
    if (((cntr - 1) * mimeBoundary) < (originalStr.Length - 1))
    {
        newStr.AppendLine(originalStr.ToString(((cntr - 1) * mimeBoundary),
            ((originalStr.Length) - ((cntr - 1) * mimeBoundary))));
    }
    return newStr.ToString();
}
```

要把编码的字符串解码为 `byte[]`，参见范例 3.2（即 3.2 节）。

3.1.4　参考

范例 3.2（即 3.2 节）；MSDN 文档中的“`Convert.ToBase64CharArray` 方法”主题。

3.2　解码base64编码的二进制数据

3.2.1　问题

你有一个 `String`，其中包含编码为 base64 的信息，例如一个位图。你需要把此数据（它可能嵌入在电子邮件消息中）从 `String` 解码为 `byte[]`，以便访问原始的二进制数据。

3.2.2 解决方案

使用 Convert 类的静态方法 Convert.FromBase64String，可以将编码的 String 解码为其对应的 byte[]。

```
static class DataTypeExtMethods
{
    public static byte[] Base64DecodeString(this string inputStr)
    {
        byte[] decodedByteArray =
                 Convert.FromBase64String(inputStr);
        return (decodedByteArray);
    }
}
```

3.2.3 讨论

Convert 类的静态方法 FromBase64String 使得对 base64 编码过的字符串进行解码成为一件很简单的事情。该方法会返回一个 byte[]，其中包含 String 解码出的元素。

如果你通过电子邮件接收到一个已经转换成字符串的文件，比如图像文件（.bmp），就可以使用如下代码将其转换回原始的位图文件。

```
// 使用3.1节中的编码方法以得到编码过的字节数组
string bmpAsString = EncodeBitmapToString(@"CSCBCover.bmp");
// 获得一个要写入的临时文件名
string bmpFile = Path.GetTempFileName() + ".bmp";

// 使用扩展方法解码图像
byte[] imageBytes = bmpAsString.Base64DecodeString();
FileStream fstrm = new FileStream(bmpFile,
                    FileMode.CreateNew, FileAccess.Write);
using (BinaryWriter writer = new BinaryWriter(fstrm))
{
    writer.Write(imageBytes);
}
```

在这段代码中，通过 3.3.3 节中的代码获得 bmpAsString 变量。imageBytes byte[] 是转换回 byte[] 的 bmpAsString String，然后将其写回到磁盘。

要将 byte[] 编码为 String，参见范例 3.1（即 3.1 节）。

3.2.4 参考

范例 3.1（即 3.1 节）；MSDN 文档中的“Convert.FromBase64CharArray 方法”主题。

3.3 把作为byte[]返回的字符串转换为字符串

3.3.1 问题

FCL 中的许多方法都返回一个 byte[]，因为它们都提供了一种字节流服务，但是一些应用

程序需要通过这些字节流服务传递字符串。下面是其中的一些方法。

```
System.Diagnostics.EventLogEntry.Data
System.IO.BinaryReader.Read
System.IO.BinaryReader.ReadBytes
System.IO.FileStream.Read
System.IO.FileStream.BeginRead
System.IO.MemoryStream // Constructor
System.IO.MemoryStream.Read
System.IO.MemoryStream.BeginRead
System.Net.Sockets.Socket.Receive
System.Net.Sockets.Socket.ReceiveFrom
System.Net.Sockets.Socket.BeginReceive
System.Net.Sockets.Socket.BeginReceiveFrom
System.Net.Sockets.NetworkStream.Read
System.Net.Sockets.NetworkStream.BeginRead
System.Security.Cryptography.CryptoStream.Read
System.Security.Cryptography.CryptoStream.BeginRead
```

在许多情况下，这个 `byte[]` 可能包含 ASCII 或 Unicode 编码的字符。你需要采用一种方式重组这个 `byte[]` 以获得原始字符串。

3.3.2 解决方案

为了把 ASCII 值的字节数组转换成完整的字符串，使用 ASCII `Encoding` 类上的 `GetString` 方法。

```
byte[] asciiCharacterArray = {128, 83, 111, 117, 114, 99, 101,
                              32, 83, 116, 114, 105, 110, 103, 128};
string asciiCharacters = Encoding.ASCII.GetString(asciiCharacterArray);
```

为了把 Unicode 值的字节数组转换成完整的字符串，使用 Unicode `Encoding` 类上的 `GetString` 方法。

```
byte[] unicodeCharacterArray = {128, 0, 83, 0, 111, 0, 117, 0, 114, 0, 99, 0,
                              101, 0, 32, 0, 83, 0, 116, 0, 114, 0, 105, 0, 110,
                          0, 103, 0, 128, 0};
string unicodeCharacters = Encoding.Unicode.GetString(unicodeCharacterArray);
```

3.3.3 讨论

`Encoding` 类的 `GetString` 方法（通过 `ASCII` 属性返回）把字节数组中包含的 7 位 ASCII 字符转换成一个字符串。对于任何大于 127（0x7F）的值，都会把它与值 127（0x7F）进行与运算，并在字符串中显示得到的字符值。例如，如果 `byte[]` 包含值 200（0xC8），则会把这个值转换为 72（0x48），并且会显示 72 对应的字符 `H`。可以在 `System.Text` 命名空间中找到 `Encoding` 类。`GetString` 方法也被重载以接受额外的参数。该方法的重载版本可以把字符串的所有或部分字符转换为 ASCII，然后把结果存储在 `byte[]` 内的指定范围中。

`GetString` 返回一个字符串，其中包含 `byte[]` 转换的 ASCII 字符。

`Encoding` 类的 `GetString` 方法（通过 `Unicode` 属性返回）把 Unicode 字符转换成 16 位的

Unicode 值。可以在 System.Text 命名空间中找到 Encoding 类。GetString 方法返回一个字符串，其中包含 byte[] 转换的 Unicode 字符。

3.3.4　参考

MSDN 文档中的“ASCIIEncoding 类”和“UnicodeEncoding 类”主题。

3.4　把字符串传递给只接受byte[]的方法

3.4.1　问题

FCL 中的许多方法接受由字符构成的 byte[]，而不是 string。下面是其中的一些方法。

```
System.Diagnostics.EventLog.WriteEntry
System.IO.BinaryWriter.Write
System.IO.FileStream.Write
System.IO.FileStream.BeginWrite
System.IO.MemoryStream.Write
System.IO.MemoryStream.BeginWrite
System.Net.Sockets.Socket.Send
System.Net.Sockets.Socket.SendTo
System.Net.Sockets.Socket.BeginSend
System.Net.Sockets.Socket.BeginSendTo
System.Net.Sockets.NetworkStream.Write
System.Net.Sockets.NetworkStream.BeginWrite
System.Security.Cryptography.CryptoStream.Write
System.Security.Cryptography.CryptoStream.BeginWrite
```

在许多情况下，可能需要把一个 string 传入上述其中一个方法或者其他某个只接受 byte[] 的方法中。你需要采用一种方式把这个字符串分解成 byte[]。

3.4.2　解决方案

要把一个 string 转换为 ASCII 值的 byte[]，可以使用 ASCII Encoding 类的 GetBytes 方法。

```
byte[] asciiCharacterArray = {128, 83, 111, 117, 114, 99, 101,
                              32, 83, 116, 114, 105, 110, 103, 128};
string asciiCharacters = Encoding.ASCII.GetString(asciiCharacterArray);

byte[] asciiBytes = Encoding.ASCII.GetBytes(asciiCharacters);
```

要把一个 string 转换成 Unicode 值的 byte[]，可以使用 Unicode Encoding 类的 GetBytes 方法。

```
byte[] unicodeCharacterArray = {128, 0, 83, 0, 111, 0, 117, 0, 114, 0, 99, 0,
                              101, 0, 32, 0, 83, 0, 116, 0, 114, 0, 105, 0, 110,
                              0, 103, 0, 128, 0};
string unicodeCharacters = Encoding.Unicode.GetString(unicodeCharacterArray);

byte[] unicodeBytes = Encoding.Unicode.GetBytes(unicodeCharacters);
```

3.4.3 讨论

Encoding 类的 GetBytes 方法（通过 ASCII 属性返回）把 ASCII 字符（包含在 char[] 或 string 中）转换成 7 位 ASCII 值的 byte[]。任何大于 127（0x7F）的值都会被转换为 ? 字符。可以在 System.Text 命名空间中找到 Encoding 类。GetBytes 方法也被重载以接受额外的参数。该方法的重载版本可以把字符串中的所有或部分字符转换成 ASCII，然后把结果存储在 byte[] 内的指定范围内，并将其返回给调用者。

Encoding 类的 GetBytes 方法（通过 Unicode 属性返回）把 Unicode 字符转换成 16 位的 Unicode 值。可以在 System.Text 命名空间中找到 Encoding 类。GetBytes 方法返回一个 byte[]，其中每个元素都包含字符串中单个字符的 Unicode 值。

源 string 或源 char[] 中的单个 Unicode 字符对应 byte[] 的两个元素。例如，下面的 byte[] 包含字母 S 的 ASCII 值。

```
byte[] sourceArray = {83};
```

不过，为了使 byte[] 包含字母 S 的 Unicode 表示，它必须包含两个元素，如下所示。

```
byte[] sourceArray2 = {83, 0};
```

Intel 体系架构使用小端（little-endian）编码方式，这意味着第一个元素是最低有效字节，第二个元素是最高有效字节。其他体系架构可能使用大端（big-endian）编码方式，与小端方式相反。UnicodeEncoding 类同时支持大端和小端编码。使用 UnicodeEncoding 实例构造函数，可以构造一个使用大端或小端排序的实例。这是通过使用下面两个构造函数之一来完成的。

```
public UnicodeEncoding (bool bigEndian, bool byteOrderMark);
public UnicodeEncoding (bool bigEndian, bool byteOrderMark,
                        bool throwOnInvalidBytes);
```

第一个参数 bigEndian 接受一个布尔参数。把这个参数设置为 true 就使用大端编码，设置为 false 则使用小端编码。

此外，你还可以选择指定是否应该生成字节顺序标记（BOM）报头，以便文件的阅读者知道使用的是大端编码还是小端编码。

3.4.4 参考

MSDN 文档中的“ASCIIEncoding 类”和“UnicodeEncoding 类”主题。

3.5 确定一个字符串是否为有效的数字

3.5.1 问题

你有一个可能包含一个数字值的字符串，你需要知道该字符串是否包含一个有效的数字。

3.5.2　解决方案

使用任意数字类型的静态 TryParse 方法。例如，要确定一个字符串是否包含一个 double，可使用下列方法。

```
string str = "12.5";
double result = 0;
if(double.TryParse(str,
        System.Globalization.NumberStyles.Float,
        System.Globalization.NumberFormatInfo.CurrentInfo,
        out result))
{
      // 是一个double
}
```

3.5.3　讨论

本范例展示了如何确定一个字符串是否只包含一个数字值。如果字符串包含一个有效数字，TryParse 方法将返回 true，而且不会遇到使用 Parse 方法时的异常。

3.5.4　参考

MSDN 文档中的“Parse”和“TryParse”主题。

3.6　舍入浮点值

3.6.1　问题

你需要将一个数字舍入为一个整数，或者舍入到指定的小数点位数。

3.6.2　解决方案

要将一个数字舍入为其最接近的整数，可使用静态 Math.Round 方法，该方法只接受一个参数。

```
int i = (int)Math.Round(2.5555); // i == 3
```

如果你需要将一个浮点值舍入到指定的小数点位数，可使用重载的静态 Math.Round 方法，它接受两个参数。

```
double dbl = Math.Round(2.5555, 2); // dbl == 2.56
```

3.6.3　讨论

Round 方法易于使用，但是用户需要了解舍入运算的工作方式。Round 方法遵循 IEEE 标准 754 的第 4 节标准，这意味着如果要舍入的数字位于两个数字之间，那么 Round 运算总舍入为偶数。下面的示例说明了这一标准。

```
double dbl1 = Math.Round(1.5); // dbl1 == 2
double dbl2 = Math.Round(2.5); // dbl2 == 2
```

我们注意到，1.5 被向上舍入到最接近的整偶数（2），而 2.5 被向下舍入到最接近的整偶数（同样是 2）。在使用 Round 方法时要牢记这一点。

> 该方法被称为银行家的舍入。之所以发明它是因为它在舍入大量具有半数的数字集合时（例如包含货币的集合）引入的偏差较小。

3.6.4 参考

MSDN 文档中的“Math 类”主题。

3.7 选择一种舍入算法

3.7.1 问题

Math.Round 方法将会将值 1.5 舍入为 2，但是使用该方法也会将值 2.5 舍入为 2。也许你希望总是舍入到较大的数字（例如，将 2.5 舍入为 3 而不是 2）。相反地，你也许希望总是舍入到较小的数字（例如将 1.5 舍入为 1）。

3.7.2 解决方案

当一个值位于两个整数中间时，可以使用静态 Math.Floor 方法始终进行向上舍入。

```
public static double RoundUp(double valueToRound) =>
    Math.Floor(valueToRound + 0.5);
```

当一个值位于两个整数中间时，可使用下列技术始终进行向下舍入。

```
public static double RoundDown(double valueToRound)
{
    double floorValue = Math.Floor(valueToRound);
    if ((valueToRound - floorValue) > .5)
        return (floorValue + 1);
    else
        return (floorValue);
}
```

3.7.3 讨论

静态 Math.Round 方法舍入到最接近的偶数。但是，有时并不希望以这种方式舍入数字。静态 Math.Floor 方法可用于允许不同方式的舍入。

本范例中用于舍入数字的方法不舍入到指定小数点位数，而是舍入到最接近的整数。

3.7.4 参考

MSDN 文档中的“Math 类”主题。

3.8 安全地执行窄化数据转换

3.8.1 问题

用户需要将一个较大的值转换为一个较小的值，同时优雅地处理转换导致的信息丢失。例如，仅当 long 数据类型大于 int.MaxSize 时，将一个 long 转换为一个 int 才会导致信息丢失。

3.8.2 解决方案

完成这一检查最简单的方法是使用 checked 关键字。下列扩展方法接受两个 long 数据类型，并且试图将它们相加。结果被放入一个 int 数据类型。如果存在一个溢出状况，那么会引发一个 OverflowException。

```
public static class DataTypeExtMethods
{
    public static int AddNarrowingChecked(this long lhs, long rhs) =>
        checked((int)(lhs + rhs));
}

// 使用了扩展方法的代码
long lhs = 34000;
long rhs = long.MaxValue;
try
{
    int result = lhs.AddNarrowingChecked(rhs);
}
catch(OverflowException)
{
    // 不能加到一起
}
```

这是最简单的方法。如果不希望有引发异常的开销并且不希望必须将大量代码封装到 try-catch 语句块中处理溢出状况，那么可以使用每种类型的 MaxValue 和 MinValue 字段。使用这些字段的检查可在转换之前进行，以确保不会发生信息丢失。如果转换将会导致信息丢失，代码可以提前通知应用程序。可以使用下列条件语句确定 sourceValue 是否能被转换成一个 short 而不会丢失任何信息。

```
// 两个变量被声明和初始化
int sourceValue = 34000;
short destinationValue = 0;

// 确定sourceValue被转换成一个short时是否会丢失任何信息：
if (sourceValue <= short.MaxValue && sourceValue >= short.MinValue)
    destinationValue = (short)sourceValue;
else
{
    // 通知应用程序将会发生信息丢失
}
```

3.8.3 讨论

窄化转换（narrowing conversion）发生在将一个较大类型转换成一个较小类型时。例如，考虑将一个 Int32 类型的值转换为一个 Int16 类型的值。如果 Int32 值小于或等于 Int16.MaxValue 字段并且 Int32 值大于或等于 Int16.MinValue 字段，那么转换进行时不会出现错误或信息丢失。信息丢失发生在当 Int32 值大于 Int16.MaxValue 字段或者 Int32 值小于 Int16.MinValue 字段的时候。在这两种情况下，Int32 最重要的位被截去并丢弃，转换后值发生了变化。

如果信息丢失发生在未检查的上下文中，它将悄悄地发生而不会通知应用程序。该问题可能导致某些非常有害且难以跟踪的错误。为了防止这一错误，检查要转换的值，确定它是否位于将要转换到的类型下限和上限范围内。如果值超出了这些范围，那么可以编写代码处理这种情况。该代码可以防止转换发生和/或通知应用程序转换问题。该解决方案有助于防止难以查找的算法错误悄悄进入用户应用程序。

3.8.2 节中所示的两种技术都是有效的。尽管如此，具体使用哪种技术将取决于预期是经常性地还是仅仅偶然地遇到溢出情况。如果预期经常遇到溢出情况，你也许会想选择第二种手动测试数值的技术，否则使用第一种技术中的 checked 关键字会比较容易。

> 在 C# 中，代码可以在已检查（checked）或未检查（unchecked）的上下文中运行。默认情况下，代码会在未检查的上下文中运行。在已检查的上下文中，任何涉及整数类型的算法和转换都会经过检查，以确定是否存在溢出状况。如果存在，则会引发一个 OverflowException。在未检查的上下文名，当溢出状况存在时不会引发 OverflowException。
>
> 通过使用 /checked{+} 编译器开关将 Check for Arithmetic Overflow/Underflow 项目属性设置为 true，或者通过使用 checked 关键字，可以建立一个已检查的上下文。通过使用 /checked- 编译器开关将 Check for Arithmetic Overflow/Underflow 项目属性设置为 false，或者通过使用 unchecked 关键字，可以建立一个未检查的上下文。

在执行转换时应当知道下列内容。

- 将一个 float、double 或 decimal 转换成一个整数类型会导致数值的小数部分被截掉。此外，

如果数值的整数部分超过了目标类型的 MaxValue，那么结果将会变得未定义，除非转换在一种已检查的上下文中进行；在这种情况下，它会触发一个 OverflowException。

- 将一个 float 或 double 强制转换成一个 decimal 会导致 float 或 double 被舍入到 28 位小数点。
- 将一个 double 强制转换成一个 float 会导致 double 被舍入到最接近的浮点值。
- 将一个 decimal 转换成一个 float 或 double 会导致 decimal 被舍入为结果类型（float 或 double）。
- 将一个 int、uint 或 long 转换成一个 float 可能导致精度丢失，但不会改变量级。
- 将一个 long 转换为一个 double 可能会导致精度丢失，但不会改变量级。

3.8.4 参考

MSDN 文档中的“checked 关键字”“Checked 和 Unchecked”主题。

3.9 测试有效的枚举值

3.9.1 问题

当向一个接收枚举值的方法传递一个数字值时，很可能会传递一个枚举中不存在的值。你希望在使用该枚举值之前执行测试，确定它是否确实是该枚举类型中定义的项。

3.9.2 解决方案

为了防止这一问题发生，可使用一个 switch 语句列出有效值，测试用户允许用于枚举类型参数的特定枚举值。

使用下列 Language 枚举。

```
public enum Language
{
    Other = 0, CSharp = 1, VBNET = 2, VB6 = 3,
    All = (Other | CSharp | VBNET | VB6)
}
```

假设有一个接收 Language 枚举的方法，例如下列方法。

```
public void HandleEnum(Language language)
{
    // 此处使用language……
}
```

你需要一个方法，定义可在 HandleEnum 中接受的枚举值。下面所示的 CheckLanguageEnumValue 方法可完成这项工作。

```
public static bool CheckLanguageEnumValue(Language language)
{
    switch (language)
    {
```

```
            // 列出枚举的所有有效类型
            // 这意味着仅指定的项是有效的
            // 而不是该枚举的所有枚举值
            case Language.CSharp:
            case Language.Other:
            case Language.VB6:
            case Language.VBNET:
                break;
            default:
                Debug.Assert(false,
                    $"{language} is not a valid enumeration value to pass.");
                return false;
        }
        return true;
    }
```

3.9.3 讨论

尽管 Enum 类包含静态 IsDefined 方法，但不应当使用它。IsDefined 内部使用反射，它会导致性能损失。同样，枚举的版本化未能得到很好的处理。考虑在软件的下一版本中将值 ManagedCPlusPlus 添加到 Language 枚举中的情况。如果 IsDefined 被用于检查此处的参数，那么它将允许 MgdCpp 作为一个合法值，因为它在枚举中定义了，尽管用于验证参数的代码未被设计用于处理它。通过明确使用 CheckLanguageEnumValue 中所示的 switch 语句，将会拒绝 MgdCpp 值，而代码不会试图在一个无效的环境下运行，因为这毕竟是用户首要追求的。

在方法对外部对象可见的情况下，应当始终进行枚举检查。一个外部对象可以调用带有公共可见性的方法，因此，任意传入该方法的枚举值都应当在其实际使用之前进行筛选。

带有私有可见性的方法可能不需要这种额外的保护级别。使用你自己的判断来决定是否使用 CheckLanguageEnumValue 方法去评估传递给私有方法的枚举值。

HandleEnum 方法能够以若干不同的方式进行调用，其中两种如下所示。

```
    HandleEnum(Language.CSharp);
    HandleEnum((Language)1); // 1是CSharp
```

任意一种方法调用都是合法的。不幸的是，下面的方法调用也是合法的。

```
    HandleEnum((Language)100);
    int someVar = 42;
    HandleEnum((Language)someVar);
```

这些方法调用也能无错误地通过编译，但是如果 HandleEnum 中的代码试图使用传递给它的值（在这种情况下，值为 100），那么将会发生古怪的行为。在许多情况下，甚至不会引发异常；HandleEnum 仅仅是接受值 100 作为参数，就如同它是 Language 枚举的合法值一样。

CheckLanguageEnumValue 方法通过筛选参数找出有效的 Language 枚举值来防止这种情况的发生。下列代码给出 HandleEnum 方法修改后的方法体。

```
    public static void HandleEnum(Language language)
    {
```

```
        if (CheckLanguageEnumValue(language))
        {
            // 此处使用language
            Console.WriteLine($"{language} is an OK enum value");
        }
        else
        {
            // 此处处理无效的枚举值
            Console.WriteLine($"{language} is not an OK enum value");
        }
    }
```

3.10 在位掩码中使用枚举成员

3.10.1 问题

你需要一个枚举，其值可以作为 bit 标志参与求或操作以创建枚举中值（标志）的组合。

3.10.2 解决方案

使用 Flags 特性标记枚举。

```
[Flags]
public enum RecycleItems
{
    None         = 0x00,
    Glass        = 0x01,
    AluminumCans = 0x02,
    MixedPaper   = 0x04,
    Newspaper    = 0x08
}
```

组合该枚举的元素只要简单地使用位或运算符（|）即可，如下所示。

```
RecycleItems items = RecycleItems.Glass | RecycleItems.Newspaper;
```

3.10.3 讨论

向枚举添加 Flags 特性标志着该枚举被视为可进行或操作的单独的位标志。使用标志的枚举与使用常规枚举类型没有任何区别。要注意即便枚举值被用作位标志，未能用 Flags 特性标记枚举也不会产生异常或者编译时错误。

添加 Flags 特性提供了两个好处。第一，如果将 Flags 特性置于一个枚举上，那么 ToString 和 ToString("G") 方法返回一个由逗号分隔的常量名所构成的字符串。否则，这两个方法返回枚举值的数字表示。要注意，ToString("F") 方法返回一个由逗号分隔的常量名所构成的字符串，而不管该枚举是否被 Flags 特性所标记。第二，当你检查代码并遇到一个枚举时，可以更好地确定开发人员对该枚举的意图。如果开发人员显式地将其定义为包含位标志（使用 Flags 特性），你就可以照此来使用它。

带有 Flags 特性的枚举可被视为一个单独的值或者由一个或多个值组合而成的单个枚举值。如果需要一次接受多种语言，可以编写下列代码。

```
RecycleItems items = RecycleItems.Glass | RecycleItems.Newspaper;
```

变量 items 现在等于两个枚举值按位求或运算后的值。这些值求或运算后等于 3，如下所示。

```
RecycleItems.Glass         0001
RecycleItems.AluminumCans  0010
ORed bit values            0011
```

枚举值被转换为二进制，进行求或运算后得到二进制值 0011 或者十进制值 3。编译器将该值视为两个单独的枚举值（RecycleItems.Glass 和 RecycleItems.AluminumCans）求或运算组合在一起或是一个单独的值（3）。

为了确定是否在一个枚举变量上打开了单个标志，可使用按位与（&）运算符，如下所示。

```
RecycleItems items = RecycleItems.Glass | RecycleItems.Newspaper;
if((items & RecycleItems.Glass) == RecycleItems.Glass)
    Console.WriteLine("The enum contains the C# enumeration value");
else
    Console.WriteLine("The enum does NOT contain the C# value");
```

这段代码将显示文本“The enum contains the C# enumeration value”。如果变量 items 未包含值 RecycleItems.Glass，那么对这两个值求与运算将产生 0。如果 items 包含该枚举值，那么将产生值 RecycleItems.Glass。实质上，以二进制格式对这两个值求与运算如下所示。

```
RecycleItems.Glass | RecycleItems.AluminumCans 0011
RecycleItems.Glass                             0001
ANDed bit values                               0001
```

我们将在范例 3.11（即 3.11 节）中详细处理这一主题。

在某些情况下，枚举可能变得相当大。可以向该枚举中添加许多其他可重复使用的数据项，如下所示。

```
[Flags]
public enum RecycleItems
{
    None         = 0x00,
    Glass        = 0x01,
    AluminumCans = 0x02,
    MixedPaper   = 0x04,
    Newspaper    = 0x08,
    TinCans      = 0x10,
    Cardboard    = 0x20,
    ClearPlastic = 0x40,
}
```

当需要一个 RecycleItems 枚举值来表示所有可重复使用的项时，用户必须对该枚举中的所有值进行求或运算。

```
RecycleItems items = RecycleItems.Glass | RecycleItems.AluminumCans |
                     RecycleItems.MixedPaper;
```

如果不这样做，你也可以简单地向枚举中添加一个包含所有可重复使用项的新值。

```
[Flags]
public enum RecycleItems
{
    None            = 0x00,
    Glass           = 0x01,
    AluminumCans    = 0x02,
    MixedPaper      = 0x04,
    Newspaper       = 0x08,
    TinCans         = 0x10,
    Cardboard       = 0x20,
    ClearPlastic    = 0x40,

    All = (None | Glass | AluminumCans | MixedPaper | Newspaper | TinCans |
           Cardboard | ClearPlastic)
}
```

现在已有了一个单独的枚举值 All，它包含着该枚举的所有值。要注意，生成 All 枚举值有两种方法。第二种方法更容易阅读。不管使用哪种方法，如果添加或删除了枚举中的单个语言元素，那么就必须相应地修改 All 值。

应当为所有枚举提供一个 None 值，即便在“以上皆非”没有意义的情况下也是如此，因为将字面量 0 赋给一个枚举总是合法的，而且因为枚举变量被赋予默认值 0 开始它的整个生命期。

同样，也可以添加值来捕获枚举值的特定子集，如下所示。

```
[Flags]
enum Language
{
    CSharp = 0x0001, VBNET = 0x0002, VB6 = 0x0004, Cpp = 0x0008,
    CobolNET = 0x000F, FortranNET = 0x0010, JSharp = 0x0020,
    MSIL = 0x0080,
    All = (CSharp | VBNET | VB6 | Cpp | FortranNET | JSharp | MSIL),
    VBOnly = (VBNET | VB6),
    NonVB = (CSharp | Cpp | FortranNET | JSharp | MSIL)
}
```

现在，该枚举中拥有了两个额外成员，一个仅包含 VB 语言（Languages.VBNET 和 Languages.VB6），另一个包含非 VB 语言。

3.11 确定是否设置了一个或多个枚举标志

3.11.1 问题

你需要确定一个由位标志构成的枚举类型变量是否包含一个或多个特定标志。例如，给定

下列枚举 Language。

```
[Flags]
enum Language
{
    CSharp = 0x0001, VBNET = 0x0002, VB6 = 0x0004, Cpp = 0x0008
}
```

使用布尔逻辑判定下列代码行中的变量 lang 是否包含 Language.CSharp 和 / 或 Language.Cpp 之类的语言。

```
Language lang = Language.CSharp | Language.VBNET;
```

3.11.2 解决方案

要确定一个变量是否包含已设置的单个位标志，可使用下列条件语句。

```
if((lang & Language.CSharp) == Language.CSharp)
{
    // lang至少包含了Language.CSharp
}
```

要确定一个变量是否仅包含已设置的单个位标志，可使用下列条件语句。

```
if(lang == Language.CSharp)
{
    // lang仅包含Language.CSharp
}
```

要确定一个变量是否包含已设置的一个位标志集合，可使用下列条件语句。

```
if((lang & (Language.CSharp | Language.VBNET)) ==
  (Language.CSharp | Language.VBNET))
{
     // lang至少包含了Language.CSharp和Language.VBNET
}
```

要确定一个变量是否只包含已设置的一个位标志集合，可使用下列条件语句。

```
if((lang | (Language.CSharp | Language.VBNET)) ==
  (Language.CSharp | Language.VBNET))
{
     // lang仅包含了Language.CSharp和Language.VBNET
}
```

3.11.3 讨论

当把枚举用作位标志并且用 Flags 特性进行标记时，它们通常会需要执行某种类型的条件测试。这些测试使位与运算符（&）和或运算符（|）成为必需。

要测试一个变量是否设置了一个特定的位标志，可使用下列条件语句来完成。

```
if((lang & Language.CSharp) == Language.CSharp)
```

其中，lang 是 Language 枚举类型。

& 运算符使用一个位掩码来确定某个位是否被设置为 1。仅当同时为 1 时，对两个位求与运算的结果才是 1；否则结果为 0。可以使用该运算符确定在包含独立位标志的数字中，某个特定位标志是否被设置为 1。如果对变量 lang 和要测试的特定位标志（此处是 Language.CSharp）求与运算，那么用户可以提取该特定位标志。如果 lang 等于 Language.CSharp，那么表达式（lang & Language.CSharp）将以下列方式求解。

```
Language.CSharp  0001
lang             0001
Anded bit values 0001
```

如果 lang 等于其他值，例如 Language.VBNET，那么表达式以下列方式求解。

```
Language.CSharp  0001
lang             0010
Anded bit values 0000
```

我们注意到，对位求与在第一个表达式中返回值 Language.CSharp，在第二个表达式中返回 0x0000。通过将该结果与待查找的值（Language.CSharp）相比较，可以看出该特定位是否已打开。

该方法对于检查特定位非常好用，但是如果希望知道是否仅有一个特定位被打开（而所有其他位都被关闭）或关闭（而所有其他位都被打开），该怎么办呢？要测试 lang 变量是否只有 Language.CSharp 位被打开，可以使用下列条件语句。

```
if(lang == Language.CSharp)
```

如果变量 lang 仅包含值 Language.CSharp，使用或运算符的表达式将如下所示。

```
lang = Language.CSharp;
if ((lang != 0) &&(Language.CSharp == (lang | Language.CSharp)))
{
    // 使用或逻辑找到了CSharp
}

Language.CSharp 0001
lang 0001
ORed bit values 0001
```

现在，向变量 lang 添加一种或两种语言并对 lang 执行相同的操作。

```
lang = Language.CSharp | Language.VB6 | Language.Cpp;
if ((lang != 0) &&(Language.CSharp == (lang | Language.CSharp)))
{
    // 使用或逻辑找到了CSharp
}

Language.CSharp 0001
lang 1101
ORed bit values 1101
```

第一个表达式的结果值与你正在测试的相同。第二个表达式的结果值大大超过了

Language.CSharp。这说明，第一个表达式的变量 lang 仅包含值 Language.CSharp，而第二个表达式除了 Language.CSharp 外还包含其他语言（也可能根本没有包含 Language.CSharp）。

使用该公式的或版本，用户可以测试多个位，以确定它们是否都被打开并且所有其他位都被关闭，如下列条件语句所示。

```
if((lang != 0) && ((lang | (Language.CSharp | Language.VBNET)) ==
   (Language.CSharp | Language.VBNET)))
```

注意，为了测试多种语言，可以简单地对语言值求或运算。通过将第一个 | 运算符转换为 & 运算符，可以确定是否至少有这些位被打开，如下列条件语句所示。

```
if((lang != 0) && ((lang & (Language.CSharp | Language.VBNET)) ==
   (Language.CSharp | Language.VBNET)))
```

当测试多个枚举值时，将想要测试的所有值求或运算的结果值添加到枚举中会带来一些好处。如果希望测试除 Language.CSharp 之外的所有其他语言，那么条件语句就会变得相当巨大且难以处理。为了解决这一问题，可以向 Language 枚举添加一个值，对除 Language.CSharp 外的所有语言求或运算。新枚举如下所示。

```
[Flags]
enum Language
{
    CSharp = 0x0001, VBNET = 0x0002, VB6 = 0x0004, Cpp = 0x0008,
    AllLanguagesExceptCSharp = VBNET | VB6 | Cpp
}
```

条件语句可能如下所示。

```
if((lang != 0) && (lang | Language.AllLanguagesExceptCSharp) ==
    Language. AllLanguagesExceptCSharp)
```

这相对短小一些，更易于管理和阅读。

> 在测试一个或多个位是否被设置为 1 时使用与运算符。在测试一个或多个位是否被设置为 0 时使用或运算符。

第 4 章

语言集成查询和lambda表达式

4.0　简介

语言集成查询（language integrated query，LINQ）是访问许多不同来源数据的卓越方式。LINQ 提供了一种可以在单个查询中分别或者同时操作不同数据领域的单一查询模型。LINQ 为 .NET 语言引入了查询数据的能力，并且其中一些语言已经提供了扩展，使得它的使用更为直观；其中一种语言就是 C#。在 C# 中有许多对该语言的扩展，有助于以一种功能丰富、直观的方式为查询提供便利。

传统的面向对象编程基于一种命令式（imperative）风格。在这种风格下，开发人员不仅要详细描述他们希望发生什么事情，还要详细准确描述如何通过代码来执行。LINQ 有助于使代码更具声明性（declarative），从而便于开发人员描述他们想要做什么，而不用详述如何达到目标。LINQ 还支持更函数式的编程风格。这些变化可以显著减少执行某些任务所需的代码量。也就是说，面向对象编程在 C# 和 .NET 中仍然具有强大的生命力，但是 C# 语言第一次提供了一种机会，让你根据自己的需要选择编程风格。不过要注意的是，LINQ 并不是对各种情况都适用，也不能代替良好的设计或实践。使用 LINQ 也可能编写出糟糕的代码，就如同可能编写出糟糕的面向对象或过程式代码一样。这其中的难点一直都在于清楚什么时候适合使用哪种技术。

LINQ 的初始版本包含许多数据领域，如下所示。

- LINQ to Object
- LINQ to XML
- LINQ to ADO.NET
- LINQ to SQL
- LINQ to DataSet

- LINQ to Entity

当你刚开始学习 LINQ 时，很容易把它看成一个新的对象关系映射层、`IEnumerable<T>` 上灵巧的新构件、一个新的 XML API 甚至只是不用再直接编写 SQL 语句的一个借口。你可以把 LINQ 当作这些来使用，但是我们鼓励你把 LINQ 视为应用程序如何请求、计算或转换来自单一源和不同源的数据集。需要花一点时间来熟悉 LINQ 的功能，但是一旦走过这一步，你就会对自己能够用它所做的事情感到吃惊。本章首先将介绍可以利用 LINQ 做什么，并且希望引导你考虑自己的哪些情况适合使用 C# 中的这种新能力。

为了编写 LINQ 查询表达式以指定条件和选择数据，我们使用 lambda 表达式。它们是表示传递给 LINQ 查询的委托的一种简便方式。例如，为了缩小结果集而调用 `Enumerable.Where` 方法时传入的 `System.Func<T, TResult>` 委托。lambda 表达式是具有不同语法的函数，这允许它们用于表达式上下文中，代替通常作为类成员的面向对象的方法。这意味着利用一种语法，就可以表达方法定义、声明以及调用委托来执行它，就像匿名方法可以做到的那样，但它的语法更简洁。*投影*（projection）是把一种类型转换成另一种类型的 lambda 表达式。

lambda 表达式如下所示。

```
j => j * 42
```

这意味着“把 `j` 用作函数的参数，`j` 最终对应 `j*42` 这个结果”。对于这个表达式以及像下面这样声明的投影来说，可以把 `=>` 读作“对应于”。

```
j => new { Number = j*42 };
```

回想一下，在 C# 1.0 中可以做同样的事情。

```
public delegate int IncreaseByANumber(int j);
public delegate int MultipleIncreaseByANumber(int j, int k, int l);

static public int MultiplyByANumber(int j)
{
    return j * 42;
}


public static void ExecuteCSharp1_0()
{
    IncreaseByANumber increase =
       new IncreaseByANumber(
           DelegatesEventsLambdaExpressions.MultiplyByANumber);
    Console.WriteLine(increase(10));
}
```

在 C# 2.0 中，利用匿名方法可以把 C# 1.0 的语法简化为以下示例，因为不再需要提供委托的名称，而且我们只想要运算的结果。

```
public delegate int IncreaseByANumber(int j);

public static void ExecuteCSharp2_0()
{
```

```
        IncreaseByANumber increase =
           new IncreaseByANumber(
            delegate(int j)
            {
                return j * 42;
            });
        Console.WriteLine(increase(10));
    }
```

这把我们带回到现在的 C# 和 lambda 表达式，如今我们只需编写如下代码。

```
    public static void ExecuteCSharp6_0()
    {
        // 声明lambda表达式
        IncreaseByANumber increase = j => j * 42;
        // 调用方法并在控制台上输出420
        Console.WriteLine(increase(10));

        MultipleIncreaseByANumber multiple = (j, k, l) => ((j * 42) / k) % l;
        Console.WriteLine(multiple(10, 11, 12));
    }
```

类型推断可以帮助编译器从 `IncreaseByANumber` 委托类型的声明推断 `j` 的类型。如果有多个参数，那么 lambda 表达式将如下所示。

```
    MultipleIncreaseByANumber multiple = (j, k, l) => ((j * 42) / k) % l;
    Console.WriteLine(multiple(10, 11, 12));
```

本章中的范例利用了委托、事件和 lambda 表达式。在其他主题中，这些范例涵盖了以下内容。

- 单独处理在多播委托中调用的每个方法
- 同步委托调用和异步委托调用
- 利用事件增强现有的类
- lambda 表达式、闭包和函数对象的多种用法

如果你不熟悉委托、事件或 lambda 表达式，就应该阅读一下关于这些主题的 MSDN 文档。还有一些很好的教程和示例代码，说明了如何以基本的方式建立和使用它们。

4.1　查询消息队列

4.1.1　问题

你希望能够以特定的条件从现有消息队列中查询消息。

4.1.2　解决方案

使用 `EnumerableMessageQueue` 类编写一个 LINQ 查询，使用 `System.Messaging.MessageQueue` 类型检索消息。

```
string queuePath = @".\private$\LINQMQ";
EnumerableMessageQueue messageQueue = null;
if (!EnumerableMessageQueue.Exists(queuePath))
    messageQueue = EnumerableMessageQueue.Create(queuePath);
else
    messageQueue = new EnumerableMessageQueue(queuePath);

using (messageQueue)
{
    BinaryMessageFormatter messageFormatter = new BinaryMessageFormatter();

    // 使用下列条件来查询消息队列中的特定消息
    // 1) label必须小于5
    // 2) 消息体中的类型名称必须包含CSharpRecipes.D
    // 3) 结果需要使用类型名称(消息体中的)降序排序

    var query = from Message msg in messageQueue
        // 对msg.Formatter的首次赋值是为了能够访问Message对象
        // 这一操作将BinaryMessageFormatter赋给了每个消息实例,
        // 以便于读取它以确定是否符合条件
        // 完成赋值操作后,通过执行相等检查以检查格式化器是否被正确赋值,
        // 该相等检查符合where语句要求的Boolean结果,
        // 同时仍然执行格式化器的赋值操作
    where ((msg.Formatter = messageFormatter) == messageFormatter) &&
                int.Parse(msg.Label) < 5 &&
                msg.Body.ToString().Contains("CSharpRecipes.D")
            orderby msg.Body.ToString() descending
            select msg;

    // 检查结果,查看label < 5并且在名称中包含CSharpRecipes.D
    foreach (var msg in query)
        Console.WriteLine($"Label: {msg.Label}" +
            $" Body: {msg.Body}");
}
```

该查询从 MessageQueue 中检索数据，通过如下条件选择消息：在这些消息中，Label 是一个小于 5 的数字，并且消息体中包含文本 CSharpRecipes.D。然后返回这些消息，并按消息体以降序对它们进行排序。

4.1.3 讨论

在这段使用 LINQ 的代码中有下面这些关键字，之前没有用于访问消息队列。

- var

  指示编译器通过语句的右边推断变量的类型。实质上，变量的类型是由隔开 var 关键字与表达式之间的运算符右边的内容确定的。这允许隐式类型化的局部变量。

- from

  from 关键字指定要查询的源集合，并创建一个范围变量来表示该集合中的单个元素。它总是查询操作中的第一个子句。如果你习惯于使用 SQL 并且期望首先看到 select，这可能是违反直觉的，但是考虑到首先需要明确操作对象而不是确定要返回什么，那么

这就是有意义的。事实上，如果我们尚未习惯 SQL 的方式，那么 SQL 的方式看起来会有些违反直觉。

- where

  where 关键字用于指定约束条件，通过它们对需要返回的元素进行筛选。每个条件的求值结果必须是一个 Boolean 值，当所有表达式都求值为 true 时，就允许选择集合的此元素。

- orderby

  orderby 指示应该根据指定的条件对结果集进行排序。默认顺序是升序，并且元素使用默认的比较器。

- select

  select 允许把集合中的整个元素、具有该元素的一部分及其他计算值的新类型构造或者数据项的子集合投影到结果中。

messageQueue 集合是 System.Messaging.MessageQueue 类型，它实现了 IEnumerable 接口。这是很重要的，因为提供的 LINQ 方法要求集合至少实现 IEnumerable，以便使之处理这个集合。实现一组不需要 IEnumerable 的扩展方法是可能的，但是大多数人没有这样的需求。当集合实现 IEnumerable<T> 时甚至更好一些，因为这样 LINQ 就会知道它正在处理的集合中的元素类型。

即使 MessageQueue 实现了 IEnumerable 接口（但没有实现 IEnumerable<T>），IEnumerable 的原始实现有一些问题，所以现在如果你尝试使用它，它实际上并不会枚举任何结果。如果你尝试使用 MessageQueue 上的 GetEnumerator，也将得到弃用警告：“此方法返回一个错误地实现 RemoveCurrent 方法族的 MessageEnumerator 对象。请使用 GetMessageEnumerator2 代替。”

为了解决这个问题，我们创建了 EnumerableMessageQueue，它从 MessageQueue 派生，但使用 GetMessageEnumerator2 方法来实现 IEnumerable 和 IEnumerable<Message>。所以，我们可以直接使用 MessageQueue 实例和 LINQ。

```
public class EnumerableMessageQueue : MessageQueue, IEnumerable<Message>
{
    public EnumerableMessageQueue() :
        base() { }
    public EnumerableMessageQueue(string path) : base(path) { }
    public EnumerableMessageQueue(string path, bool sharedModeDenyReceive) :
        base (path, sharedModeDenyReceive) { }
    public EnumerableMessageQueue(string path, QueueAccessMode accessMode) :
        base (path, accessMode) { }
    public EnumerableMessageQueue(string path, bool sharedModeDenyReceive,
        bool enableCache) : base (path, sharedModeDenyReceive, enableCache) { }
    public EnumerableMessageQueue(string path, bool sharedModeDenyReceive,
        bool enableCache, QueueAccessMode accessMode) :
            base (path, sharedModeDenyReceive, enableCache, accessMode) { }

    public static new EnumerableMessageQueue Create(string path) =>
        Create(path, false);
```

```
    public static new EnumerableMessageQueue Create(string path,
        bool transactional)
    {
        // 直接使用MessageQueue以确保队列存在
        if (!MessageQueue.Exists(path))
            MessageQueue.Create(path, transactional);
        // 一旦确定其存在后,就创建可枚举的队列
        return new EnumerableMessageQueue(path);
    }

    public new MessageEnumerator GetMessageEnumerator()
    {
        throw new NotSupportedException("Please use GetEnumerator");
    }

    public new MessageEnumerator GetMessageEnumerator2()
    {
        throw new NotSupportedException("Please use GetEnumerator");
    }

    IEnumerator<Message> IEnumerable<Message>.GetEnumerator()
    {
        // 注意,在.NET 3.5中,你能够通过正常的LINQ语义
        // 调用MessageQueue上的"GetEnumerator"并使之工作
        // 现在我们必须调用GetMessageEnumerator2,因为GetEnumerator已被弃用
        // 现在,我们使用EnumerableMessageQueue来处理此问题
        MessageEnumerator messageEnumerator = base.GetMessageEnumerator2();
        while (messageEnumerator.MoveNext())
        {
            yield return messageEnumerator.Current;
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        // 注意,在.NET 3.5中,你能够通过正常的LINQ语义
        // 调用MessageQueue上的"GetEnumerator"并使之工作
        // 现在我们必须调用GetMessageEnumerator2,因为GetEnumerator已被弃用
        // 现在,我们使用EnumerableMessageQueue来处理此问题
        MessageEnumerator messageEnumerator = base.GetMessageEnumerator2();
        while (messageEnumerator.MoveNext())
        {
            yield return messageEnumerator.Current;
        }
    }
}
```

现在查询提供了元素类型 `Message`，如同 LINQ 查询的 `from` 行所示。

```
var query = from Message msg in messageQueue
```

在解决方案中，已经使用 `BinaryFormatter` 发送了队列中的消息。为了能够正确地查询它们，在把每个 `Message` 作为 `where` 子句的一部分进行检查之前，必须设置其上的 `Formatter` 属性。

```
// 对msg.Formatter的首次赋值是为了能够访问Message对象
// 这一操作将BinaryMessageFormatter赋给了每个消息实例,
// 以便于读取它以确定是否符合条件
// 完成赋值操作后,检查格式化器是否被正确赋值,
// 通过执行相等检查以符合where语句要求的Boolean结果,
// 同时仍然执行格式化器的赋值操作
where ((msg.Formatter = messageFormatter) == messageFormatter) &&
```

在解决方案代码中使用了两次 var 关键字，如下所示。

```
var query = from Message msg in messageQueue
            ...

foreach (var msg in query)
...
```

第一次使用 var 意味着将返回一个 IEnumerable<Message>，并被赋予 query 变量。第二次使用 var 意味着 msg 的类型是 Message，因为查询变量是 IEnumerable<Message> 类型，并且 msg 变量是该 IEnumerable 中的一个元素。

另外值得注意的是，在查询中执行运算时，可以使用实际的 C# 代码来确定条件，并且不仅仅可以使用预先确定的运算符集合。在这个查询的 where 子句中，int.Parse 和 string.Contains 都用于帮助筛选消息。

```
int.Parse(msg.Label) < 5 &&
msg.Body.ToString().Contains('CSharpRecipes.D')
```

最后，使用 orderby 对结果进行降序排序。

```
orderby msg.Body.ToString() descending
```

4.1.4　参考

范例 4.9（即 4.9 节）；MSDN 文档中的“MessageQueue 类”“隐式类型的居部变量”“from 关键字”“where 关键字”“orderby 关键字”和“select 关键字”主题。

4.2　对数据使用集合语义

4.2.1　问题

你想要使用集合操作处理集合，执行并、交、排除和去除重复项等操作。

4.2.2　解决方案

使用下列集合运算符（它们是作为标准查询运算符的一部分提供的）执行以上操作。

- Distinct

```
IEnumerable<string> whoLoggedIn =
    dailySecurityLog.Where(
        logEntry => logEntry.Contains("logged in")).Distinct();
```

- Union

```
// 并集
Console.WriteLine("Employees for all projects");
var allProjectEmployees = project1.Union(project2.Union(project3));
```

- Intersect

```
// 交集
Console.WriteLine("Employees on every project");
var everyProjectEmployees = project1.Intersect(project2.Intersect(project3));
```

- Except

```
Console.WriteLine("Employees on only one project");
var onlyProjectEmployees = allProjectEmployees.Except(unionIntersect);
```

4.2.3 讨论

标准查询运算符是表示 LINQ 模式的方法集合。这个集合包括用于执行不同类型操作的运算符，比如筛选、投影、排序、分组等，也包括集合操作。

标准查询运算符中的集合运算符包括以下这些。

- Distinct
- Union
- Intersect
- Except

Distinct 运算符用于从将要处理的集合或结果集中提取所有的非重复项。例如，假定我们有一个字符串集合，代表今天在公用开发环境中一台虚拟机上的登录和注销行为。

```
string[] dailySecurityLog = {
        "Rakshit logged in",
        "Aaron logged in",
        "Rakshit logged out",
        "Ken logged in",
        "Rakshit logged in",
        "Mahesh logged in",
        "Jesse logged in",
        "Jason logged in",
        "Josh logged in",
        "Melissa logged in",
        "Rakshit logged out",
        "Mary-Ellen logged out",
        "Mahesh logged in",
        "Alex logged in",
        "Scott logged in",
        "Aaron logged out",
        "Jesse logged out",
        "Scott logged out",
        "Dave logged in",
        "Ken logged out",
        "Alex logged out",
        "Rakshit logged in",
```

```
        "Dave logged out",
        "Josh logged out",
        "Jason logged out"};
```

从此集合中，我们想要确定今天登录到虚拟机的人员列表。由于人们可以在一天中登录和注销许多次或者一整天保持登录，我们需要消除重复的登录条目。Distinct 是 System.Linq.Enumerable 类（它实现了标准查询运算符）上的一个扩展方法，可以在字符串数组（它支持 IEnumerable）上调用它，用于从集合中获取不同项目的集合。（有关扩展方法的更多信息，参见范例 4.5，即 4.5 节。）要获得目标集合，使用另一个标准查询运算符 where，其传入一个 lambda 表达式，用于确定集合的筛选条件并检查 IEnumerable<string> 中的每个字符串，以确定字符串是否包含"logged in"。lambda 表达式是内联语句（类似于匿名方法），可用于代替委托。（有关 lambda 表达式的更多信息，参见范例 4.12，即 4.12 节。）如果字符串包含"logged in"，就选择它们。Distinct 进一步缩小了字符串集合的范围，消除了重复的"logged in"记录，只为每个用户保留一条记录。

```
IEnumerable<string> whoLoggedIn =
    dailySecurityLog.Where(
        logEntry => logEntry.Contains("logged in")).Distinct();
Console.WriteLine("Everyone who logged in today:");
foreach (string who in whoLoggedIn)
    Console.WriteLine(who);
```

为了使事情更有趣一些，对于余下的运算符，我们将处理一家公司内不同项目上的雇员集合。Employee 是一个十分简单的类，它包含一个 Name 属性，并且重写了 ToString、Equals 和 GetHashCode，如下所示。

```
public class Employee
{
    public string Name { get; set; }
    public override string ToString() => this.Name;
    public override bool Equals(object obj) =>
        this.GetHashCode().Equals(obj.GetHashCode());
    public override int GetHashCode() => this.Name.GetHashCode();
}
```

你可能想知道为什么要为这样一个简单的类重载 Equals 和 GetHashCode。原因是，当 LINQ 对集合中的元素执行比较时，它会使用默认的比较规则，这反过来又会使用 Equals 和 GetHashCode 来确定引用类型的一个实例是否与另一个实例相同。如果你没有在引用类型的类中提供语义，用以确定对象的两个实例的数据相同时它们将具有相同的散列代码或相等值，那么默认情况下这两个实例将是不同的，因为两个引用类型默认情况下具有不同的散列代码。我们重写了该方法，使得如果每个 Employee 的 Name 相同，散列代码和相等值都将正确地把实例标识为相同的。集合运算符的重载方法也接受自定义的比较器，因此也可以使你甚至能够为无法修改 Equals 和 GetHashCode 方法的类提供比较语义。

在完成这项任务后，现在就可以把 Employee 指派给各个项目。

```
Employee[] project1 = {
          new Employee(){ Name = "Rakshit" },
          new Employee(){ Name = "Jason" },
          new Employee(){ Name = "Josh" },
```

```
            new Employee(){ Name = "Melissa" },
            new Employee(){ Name = "Aaron" },
            new Employee() { Name = "Dave" },
            new Employee() {Name = "Alex" } };
Employee[] project2 = {
            new Employee(){ Name = "Mahesh" },
            new Employee() {Name = "Ken" },
            new Employee() {Name = "Jesse" },
            new Employee(){ Name = "Melissa" },
            new Employee(){ Name = "Aaron" },
            new Employee(){ Name = "Alex" },
            new Employee(){ Name = "Mary-Ellen" } };
Employee[] project3 = {
            new Employee(){ Name = "Mike" },
            new Employee(){ Name = "Scott" },
            new Employee(){ Name = "Melissa" },
            new Employee(){ Name = "Aaron" },
            new Employee(){ Name = "Alex" },
            new Employee(){ Name = "Jon" } };
```

为了查找所有项目上的所有 Employee，可使用 Union 获取全部三个项目中的所有非重复的 Employee 并输出它们，因为 Union 返回全部三个项目中的所有非重复的 Employee。

```
// 并集
Console.WriteLine("Employees for all projects");
var allProjectEmployees = project1.Union(project2.Union(project3));
foreach (Employee employee in allProjectEmployees)
    Console.WriteLine(employee);
```

然后，我们可以使用 Intersect 来获得参与了每个项目的雇员，因为 Intersect 将确定每个项目中的公共 Employee 并返回它们。

```
// 交集
Console.WriteLine("Employees on every project");
var everyProjectEmployees = project1.Intersect(project2.Intersect(project3));
foreach (Employee employee in everyProjectEmployees)
    Console.WriteLine(employee);
```

最后，我们可以结合使用 Union 和 Except，找出只在一个项目中的 Employee，因为 Except 筛选了参与不止一个项目的 Employee。

```
// 排除
var intersect1_3 = project1.Intersect(project3);
var intersect1_2 = project1.Intersect(project2);
var intersect2_3 = project2.Intersect(project3);
var unionIntersect = intersect1_2.Union(intersect1_3).Union(intersect2_3);

Console.WriteLine("Employees on only one project");
var onlyProjectEmployees = allProjectEmployees.Except(unionIntersect);
foreach (Employee employee in onlyProjectEmployees)
    Console.WriteLine(employee);
```

代码的输出如下所示。

```
Everyone who logged in today:
Rakshit logged in
```

```
Aaron logged in
Ken logged in
Mahesh logged in
Jesse logged in
Jason logged in
Josh logged in
Melissa logged in
Alex logged in
Scott logged in
Dave logged in
Employees for all projects
Rakshit
Jason
Josh
Melissa
Aaron
Dave
Alex
Mahesh
Ken
Jesse
Mary-Ellen
Mike
Scott
Jon
Employees on every project
Melissa
Aaron
Alex
Employees on only one project
Rakshit
Jason
Josh
Dave
Mahesh
Ken
Jesse
Mary-Ellen
Mike
Scott
Jon
```

4.2.4 参考

MSDN 文档中的“标准查询运算符”“`Distinct` 方法”“`Union` 方法”“`Intersect` 方法”和“`Except` 方法”主题。

4.3 利用LINQ to SQL重用参数化查询

4.3.1 问题

你需要利用不同的参数值多次执行相同的参数化查询，但是你想避免每次执行查询时解析

查询表达式树构建参数化 SQL 的开销。

4.3.2 解决方案

使用 CompiledQuery.Compile 方法构建一个表达式树，每次利用新参数执行查询时将不必解析它。

```
var GetEmployees =
    CompiledQuery.Compile((NorthwindLinq2Sql.NorthwindLinq2SqlDataContext nwdc,
        string ac, string ttl) =>
            from employee in nwdc.Employees
            where employee.HomePhone.Contains(ac) &&
                    employee.Title == ttl
            select employee);

var northwindDataContext = new NorthwindLinq2Sql.NorthwindLinq2SqlDataContext();
```

查询第一次执行发生在实际编译它时（在 foreach 循环中第一次调用 GetEmployees 时）。本次循环中所有其他迭代及下一次循环中都将使用编译过的版本，从而避免解析表达式树。

```
foreach (var employee in GetEmployees(northwindDataContext, "(206)",
    "Sales Representative"))
    Console.WriteLine($"{employee.FirstName} {employee.LastName}");

foreach (var employee in GetEmployees(northwindDataContext, "(71)",
    "Sales Manager"))
    Console.WriteLine($"{employee.FirstName} {employee.LastName}");
```

4.3.3 讨论

我们把 var 用于查询声明，因为它更清晰，但是在这种情况下 var 实际上是：

```
System.Func<NorthwindLinq2Sql.NorthwindLinq2SqlDataContext, string, string,
System.Linq.IQueryable<NorthwindLinq2Sql.Employee>>
```

它是我们所创建包含查询的 lambda 表达式的委托签名。是的，所有这些疯狂的查询，我们只不过实例化了一个委托。公平地说，Func 委托是作为 LINQ 的一部分添加到 System 命名空间中。因此，不要沮丧，我们仍将做一些很酷的新事情！

这说明我们将不会基于 Compile 中的结果集返回一个 IEnumerable 或 IQueryable，而是返回一个表达式树；它代表了查询的潜力，而不是查询本身。一旦有了这个树，LINQ to SQL 就必须把它转换成可用于操纵数据库的实际 SQL 语句。十分有趣的是，如果置入一个 string.Format 的调用作为检测家庭电话号码中区号的部分，我们将会得到一个 NotSupportedException，它告知我们不能把 string.Format 转换成 SQL。

```
    where employee.HomePhone.Contains(string.Format($"({ac})")) &&

System.NotSupportedException:
Method 'System.String Format(System.String,System.Object)'
  has no supported translation to SQL.
```

这是可以理解的，因为 SQL 没有 .NET Framework 方法用于执行操作的概念，但是在设计查询时要记住这是使用 LINQ to SQL 的限制。

在第一次执行之后就编译了查询，从此之后在每次迭代中都无需付出表达式树转换成参数化 SQL 的转换成本。

建议为大量使用的参数化查询编译查询，但是如果查询很少使用，可能不值得这样做。通常的做法是，剖析代码找出这样做有益的领域。

请注意，在 Entity Framework 5 及以上的模板里，不能在生成的上文中使用 `CompiledQuery`，因为这些模板重写以用于 `DbContext`，而不是 `ObjectContext`；而 `CompiledQuery.Compile` 需要 `ObjectContext`。好消息是，如果你正在使用 Entity Framework 5 及更高版本，`DbContext` 已经为你预编译了查询。你仍然可以在 LINQ to SQL 数据上下文中使用 `CompiledQuery`。

Microsoft 建议在新的开发项目中使用 `DbContext`，但是如果你有使用之前的数据访问机制的代码，`CompiledQuery` 仍然可以帮助你。

4.3.4 参考

MSDN 文档中的“`CompiledQuery.Compile` 方法”和“表达式树”主题。

4.4 以文化敏感的方式对结果排序

4.4.1 问题

你想确保在查询中排序时针对的是应用程序特定的文化，而它可能与该线程的当前文化不同。

4.4.2 解决方案

使用 `OrderBy` 查询运算符的重载版本，它接受一个自定义的比较器，用以指定要执行比较的文化。

```
// 在丹麦创建一个丹麦语的CultureInfo
CultureInfo danish = new CultureInfo("da-DK");
// 在美国创建一个英语的CultureInfo
CultureInfo american = new CultureInfo("en-US");

CultureStringComparer comparer =
   new CultureStringComparer(danish,CompareOptions.None);
var query = names.OrderBy(n => n, comparer);
```

4.4.3 讨论

如果线程的当前文化是你想使用的文化，那么在 .NET 中处理像对特定文化进行排序这样的本地化问题就是一种相对简单的任务。在 C# 中，通过包括 `System.Globalization` 命名空间来访问那些帮助处理文化问题的 Framework 类。为了使解决方案中的代码运行，就要

包括这个命名空间。一个不使用线程当前文化的示例是，在设置为美式英语的 Windows 版本上运行的应用程序需要显示丹麦语中单词的有序列表。这一功能在使用一个多租户 Web 服务或一个有着全球客户的网站时也很有用。

应用程序中的当前线程可能有用于美式英语（en-US）的 CultureInfo。默认情况下，OrderBy 的排序顺序将使用当前文化的排序设置。为了指定这个列表应该根据丹麦语规则进行排序，需要一小点工作来自定义比较器。

```
CultureStringComparer comparer =
    new CultureStringComparer(danish,CompareOptions.None);
```

comparer 变量是自定义比较器类 CultureStringComparer 的一个实例，这个类被定义为实现专用于字符串的 IComparer<T> 接口。该类用于为排序顺序提供文化设置。

```
public class CultureStringComparer : IComparer<string>
{
    private CultureStringComparer()
    {
    }

    public CultureStringComparer(CultureInfo cultureInfo, CompareOptions options)
    {
        if (cultureInfo == null)
            throw new ArgumentNullException(nameof(cultureInfo));

        CurrentCultureInfo = cultureInfo;
        Options = options;
    }

    public int Compare(string x, string y) =>
        CurrentCultureInfo.CompareInfo.Compare(x, y, Options);

    public CultureInfo CurrentCultureInfo { get; set; }

    public CompareOptions Options { get; set; }
}
```

为了演示如何使用它，首先编译一份要排序的单词列表。由于丹麦语把字符 Æ 视为一个单独的字母，在字母表中将其排在 Z 后面，而英语则把字符 Æ 视为一个特殊符号，在字母表中将其排在 A 之前，这个例子将演示排序的差别。

```
string[] names = { "Jello", "Apple", "Bar", "Æble",
    "Forsooth", "Orange", "Zanzibar" };
```

现在，我们可以为丹麦语和美式英语设置 CultureInfo，并利用每种文化特有的比较规则调用 OrderBy。这个查询将不会使用查询表达式语法，而是使用 IEnumerable<string>.OrderBy() 的函数式风格。

```
// 在丹麦创建一个丹麦语的CultureInfo
CultureInfo danish = new CultureInfo("da-DK");

// 在美国创建一个英语的CultureInfo
CultureInfo american = new CultureInfo("en-US");
```

```
CultureStringComparer comparer =
    new CultureStringComparer(danish,CompareOptions.None);
var query = names.OrderBy(n => n, comparer);
Console.WriteLine($"Ordered by specific culture : " +
    $"{comparer.CurrentCultureInfo.Name}");
foreach (string name in query)
    Console.WriteLine(name);

comparer.CurrentCultureInfo = american;
query = names.OrderBy(n => n, comparer);
Console.WriteLine($"Ordered by specific culture : " +
    $"{comparer.CurrentCultureInfo.Name}");
foreach (string name in query)
    Console.WriteLine(name);

query = from n in names
        orderby n
        select n;
Console.WriteLine("Ordered by Thread.CurrentThread.CurrentCulture : " +
    $"{ Thread.CurrentThread.CurrentCulture.Name}");
foreach (string name in query)
    Console.WriteLine(name);


// 在丹麦创建一个丹麦语的 CultureInfo
    CultureInfo danish = new CultureInfo("da-DK");
    // 在美国创建一个英语的 CultureInfo
    CultureInfo american = new CultureInfo("en-US");

    CultureStringComparer comparer =
        new CultureStringComparer(danish, CompareOptions.None);
    var query = names.OrderBy(n => n, comparer);
    Console.WriteLine("Ordered by specific culture : " +
        comparer.CurrentCultureInfo.Name);
    foreach (string name in query)
    {
        Console.WriteLine(name);
    }
    comparer.CurrentCultureInfo = american;
    query = names.OrderBy(n => n, comparer);
    Console.WriteLine("Ordered by specific culture : " +
        comparer.CurrentCultureInfo.Name);
    foreach (string name in query)
    {
        Console.WriteLine(name);
    }
```

下面的输出结果显示单词 `Æble` 在丹麦语列表中出现在最后面，而在美式英语列表中则出现在最前面。

```
Ordered by specific culture : da-DK
Apple
Bar
Forsooth
```

```
Jello
Orange
Zanzibar
Æble
Ordered by specific culture : en-US
Æble
Apple
Bar
Forsooth
Jello
Orange
Zanzibar
```

4.4.4 参考

MSDN 文档中的“OrderBy 方法”“CultureInfo 类”和“IComparer<T> 接口”主题。

4.5 添加用于LINQ的函数式扩展

4.5.1 问题

你将对集合频繁地执行一些操作，这些操作目前位于实用程序类中。你希望能够以一种更加无缝的方式对集合使用这些操作，而不必把对集合的引用传递给实用程序类。

4.5.2 解决方案

使用扩展方法帮助实现更加偏向函数式编程风格的集合操作。例如，为了向数字集合中添加一个加权的移动平均计算操作，可以在一个静态类中实现一组 WeightedMovingAverage 扩展方法，然后将它们作为这些集合的一部分进行调用。

```
decimal[] prices = new decimal[10] { 13.5M, 17.8M, 92.3M, 0.1M, 15.7M,
                                     19.99M, 9.08M, 6.33M, 2.1M, 14.88M };
Console.WriteLine(prices.WeightedMovingAverage());

double[] dprices = new double[10] { 13.5, 17.8, 92.3, 0.1, 15.7,
                                    19.99, 9.08, 6.33, 2.1, 14.88 };
Console.WriteLine(dprices.WeightedMovingAverage());

float[] fprices = new float[10] { 13.5F, 17.8F, 92.3F, 0.1F, 15.7F,
                                  19.99F, 9.08F, 6.33F, 2.1F, 14.88F };
Console.WriteLine(fprices.WeightedMovingAverage());

int[] iprices = new int[10] { 13, 17, 92, 0, 15,
                              19, 9, 6, 2, 14 };
Console.WriteLine(iprices.WeightedMovingAverage());

long[] lprices = new long[10] { 13, 17, 92, 0, 15,
                                19, 9, 6, 2, 14 };
Console.WriteLine(lprices.WeightedMovingAverage());
```

为了给所有数字类型提供 WeightedMovingAverage，在 LinqExtensions 类中提供了用于可空数字类型和非空数字类型的方法。

```
public static class LinqExtensions
{
    public static decimal? WeightedMovingAverage(
        this IEnumerable<decimal?> source)
    {
        if (source == null)
            throw new ArgumentNullException(nameof(source));

        decimal aggregate = 0.0M;
        decimal weight;
        int item = 1;
        // 对非空的数据项计数,将其作为加权的因子
        int count = source.Count(val => val.HasValue);
        foreach (var nullable in source)
        {
            if (nullable.HasValue)
            {
                weight = item / count;
                aggregate += nullable.GetValueOrDefault() * weight;
                count++;
            }
        }
        if (count > 0)
            return new decimal?(aggregate / count);
        return null;
    }
    // 接下来是跟如上方法相同的方法模式,用于每个类型和
    // 它对应的可空类型(double / double?、int / int?等)
}
```

4.5.3　讨论

扩展方法允许你创建看起来属于集合一部分的操作。它们都是静态方法，可以像实例方法一样调用它们，从而允许你扩展现有类型。必须在未嵌套的静态类中声明扩展方法。一旦定义了带有扩展方法的静态类，用于类的命名空间的 using 指令就可以在源文件中使用这些扩展。

> 如果某个实例方法具有与扩展方法相同的签名，将永远不会调用扩展方法。有冲突的扩展方法声明将解析成最近的封闭命名空间中的方法。

你不能使用扩展方法创建以下各项。

- 属性（get 和 set 方法）
- 运算符（+、- 和 = 等）
- 事件

要声明一个扩展方法，在方法声明的第一个参数前面指定 this 关键字，并且使该参数的类型是要扩展的类型。例如，在 WeightedMovingAverage 方法的 Nullable<decimal> 版本中，将支持那些支持 IEnumerable<decimal?>（或 IEnumerable<Nullable<decimal>>）的集合。

```
public static decimal? WeightedMovingAverage(this IEnumerable<decimal?> source)
{
    if (source == null)
        throw new ArgumentNullException(nameof(source));

    decimal aggregate = 0.0M;
    decimal weight;
    int item = 1;
    // 对非空的数据项计数,将其作为加权的因子
    int count = source.Count(val => val.HasValue);
    foreach (var nullable in source)
    {
        if (nullable.HasValue)
        {
            weight = item / count;
            aggregate += nullable.GetValueOrDefault() * weight;
            count++;
        }
    }
    if (count > 0)
        return new decimal?(aggregate / count);
    return null;
}
```

System.Linq.Extensions 类中的扩展方法支持大量的 LINQ 功能，包括 Average 方法。Average 方法具有大多数数字类型，但是没有提供 short（Int16）的重载。我们自己可以轻松地为 short 和 Nullable<short> 添加重载来改正这一点。

```
public static double? Average(this IEnumerable<short?> source)
{
    if (source == null)
        throw new ArgumentNullException(nameof(source));

    double aggregate = 0.0;
    int count = 0;
    foreach (var nullable in source)
    {
        if (nullable.HasValue)
        {
            aggregate += nullable.GetValueOrDefault();
            count++;
        }
    }
    if (count > 0)
        return new double?(aggregate / count);
    return null;
}

public static double Average(this IEnumerable<short> source)
{
```

```
        if (source == null)
            throw new ArgumentNullException(nameof(source));

        double aggregate = 0.0;
        // 使用数据源中的数据项计数
        int count = source.Count();
        foreach (var value in source)
        {
            aggregate += value;
        }
        if (count > 0)
            return aggregate / count;
        else
            return 0.0;
    }

    public static double? Average<TSource>(this IEnumerable<TSource> source,
        Func<TSource, short?> selector) =>
            source.Select<TSource, short?>(selector).Average();

    public static double Average<TSource>(this IEnumerable<TSource> source,
        Func<TSource, short> selector) =>
            source.Select<TSource, short>(selector).Average();

    #endregion // 扩展Average
```

然后，我们可以在基于 short 的集合上调用 Average 方法，就像 WeightedMovingAverage 一样。

```
    short[] sprices = new short[10] { 13, 17, 92, 0, 15, 19, 9, 6, 2, 14 };
    Console.WriteLine(sprices.WeightedMovingAverage());
    // System.Linq.Extensions没有为short实现Average,但我们添加了支持
    Console.WriteLine(sprices.Average());
```

4.5.4 参考

MSDN 文档中的“扩展方法”主题。

4.6 跨数据库执行查询和联接

4.6.1 问题

你有来自不同数据区域的两个数据集，希望能够组合这些数据并处理它们。

4.6.2 解决方案

使用 LINQ 桥接异种数据域。LINQ 被设计成以相同的使用方式跨不同数据域，并且支持使用联接语法组合这些数据集。

为了演示这一点，我们将把一个全部由类别数据组成的 XML 和来自一个数据库

（Northwind）中的产品数据进行联接，以创建产品信息的一组新数据，其中包含产品名称、类别描述和类别名称。

```
Northwind dataContext =
    new Northwind(Settings.Default.NorthwindConnectionString);
ProductsTableAdapter adapter = new ProductsTableAdapter();
Products products = new Products();
adapter.Fill(products._Products);

XElement xmlCategories = XElement.Load("Categories.xml");

var expr = from product in products._Products
           where product.Units_In_Stock > 100
           join xc in xmlCategories.Elements("Category")
           on product.Category_ID equals int.Parse(
               xc.Attribute("CategoryID").Value)
           select new
           {
               ProductName = product.Product_Name,
               Category = xc.Attribute("CategoryName").Value,
               CategoryDescription = xc.Attribute("Description").Value
           };

foreach (var productInfo in expr)
{
    Console.WriteLine("ProductName: " + productInfo.ProductName +
        " Category: " + productInfo.Category +
        " Category Description: " + productInfo.CategoryDescription);
}
```

新的数据集将输出到控制台，但也可以轻松地把它重传给另一个方法，在另一个查询中转换它，或者把它写成第三种数据格式。

```
ProductName: Grandma's Boysenberry Spread Category: Condiments Category
Description: Sweet and savory sauces, relishes, spreads, and seasonings
ProductName: Gustaf's Knäckebröd Category: Grains/Cereals Category Description:
Breads, crackers, pasta, and cereal
ProductName: Geitost Category: Dairy Products Category Description:
Cheeses
ProductName: Sasquatch Ale Category: Beverages Category Description: Soft drinks,
coffees, teas, beer, and ale
ProductName: Inlagd Sill Category: Seafood Category Description:
Seaweed and fish
ProductName: Boston Crab Meat Category: Seafood Category Description:
Seaweed and fish
ProductName: Pâté chinois Category: Meat/Poultry Category Description:
Prepared meats
ProductName: Sirop d'érable Category: Condiments Category Description:
Sweet and savory sauces, relishes, spreads, and seasonings
ProductName: Röd Kaviar Category: Seafood Category Description:
Seaweed and fish
ProductName: Rhönbräu Klosterbier Category: Beverages Category Description:
Soft drinks, coffees, teas, beer, and ale
```

4.6.3 讨论

这个解决方案组合了来自两个不同数据域的数据：XML 和 SQL 数据库。在 LINQ 出现之前，为了执行该任务，不仅需要手动创建第三个数据库用于保存结果，而且需要为每个数据域编写特定的代码以查询该数据域，从而获取其中的数据（XPath 用于 XML；SQL 用于数据库），然后手动把每个数据域中的结果集转移到新的数据库中。LINQ 允许编写查询来组合两个数据集，通过投影一种新的匿名类型自动构造一种类型，并把相关的数据置于新构建的类型中，所有这些操作都采用相同的语法。这不仅可以简化代码，而且允许你把注意力更多地集中于获取你想要的数据，而较少关注如何准确地读取两个数据域。

这个示例使用 LINQ to DataSet 和 LINQ to XML 访问多个数据域。

```
var dataContext = new NorthwindLinq2Sql.NorthwindLinq2SqlDataContext();

ProductsTableAdapter adapter = new ProductsTableAdapter();
Products products = new Products();
adapter.Fill(products._Products);

XElement xmlCategories = XElement.Load("Categories.xml");
```

`NorthwindLinq2SqlDataContext` 是一个 `DataContext` 类。`DataContext` 类似于一个融为一体的 ADO.NET `Connection` 和 `Command` 对象。你使用它来建立连接、执行查询或者通过实体类直接访问表。可以通过在 Visual Studio 中添加一个新的“LINQ to SQL 类”项直接从数据库生成 `DataContext`。这允许查询访问本地 Northwind.mdf 数据库。从 Northwind.mdf 数据库中的 Products 表加载 Products `DataSet`，以便在查询中使用它。

`XElement` 是 LINQ to XML 中的主要类之一。它允许加载现有的 XML、创建新的 XML，或者通过 `ToString` 检索元素的 XML 文本。例 4-1 显示了将要加载的 Categories.xml 文件。有关 `XElement` 和 LINQ to XML 的更多信息，参见第 10 章。

例 4-1：Categories.xml

```
<?xml version="1.0" encoding="utf-8"?>
<Categories>
  <Category Id="1" Name="Beverages"
    Description="Soft drinks, coffees, teas, beers, and ales" />
  <Category Id="2" Name="Condiments"
    Description="Sweet and savory sauces, relishes, spreads, and seasonings" />
  <Category Id="3" Name="Confections"
    Description="Desserts, candies, and sweet breads" />
  <Category Id="4" Name="Dairy Products" Description="Cheeses" />
  <Category Id="5" Name="Grains/Cereals"
    Description="Breads, crackers, pasta, and cereal" />
  <Category Id="6" Name="Meat/Poultry" Description="Prepared meats" />
  <Category Id="7" Name="Produce" Description="Dried fruit and bean curd" />
  <Category Id="8" Name="Seafood" Description="Seaweed and fish" />
</Categories>
```

使用 LINQ（确切地讲，是使用 `join` 关键字）联接两个数据集。通过匹配产品表中的类别 ID 与 XML 文件中的类别 ID 来联接数据，从而组合数据。在 SQL 术语中，`join` 关键字表示一个内联接。

```
var expr = from product in products._Products
            where product.UnitsInStock > 100
            join xc in xmlCategories.Elements("Category")
            on product.CategoryID equals int.Parse(xc.Attribute("Id").Value)
```

一旦联接结果完成，使用 select 关键字投影一个新类型。

```
select new
{
    ProductName = product.ProductName,
    Category = xc.Attribute("Name").Value,
    CategoryDescription = xc.Attribute("Description").Value
};
```

这允许我们组合两个数据集中的不同数据元素产生第三个数据集，它看起来可能与原来的两个数据集完全不同。

对数据库的两个数据集执行联接是一个糟糕的主意，因为数据库可以为这些数据集更快地执行这种操作，但是当你需要联接异种数据集时，LINQ 可以提供帮助。

4.6.4 参考

MSDN 文档中的“join 关键字”“System.Data.Linq.DataContext 类”和“XElement 类”主题。

4.7 利用LINQ查询配置文件

4.7.1 问题

数据集可以存储在许多不同的位置，比如配置文件。你希望能够查询配置文件以获取信息集合。

4.7.2 解决方案

使用 LINQ 来查询配置节（configuration section）。在下面的示例中，我们从包含章节信息的自定义配置节中检索所有包含 and 的偶数章标题。

```
CSharpRecipesConfigurationSection recipeConfig =
    ConfigurationManager.GetSection("CSharpRecipesConfiguration") as
CSharpRecipesConfigurationSection;

var expr = from ChapterConfigurationElement chapter in
    recipeConfig.Chapters.OfType<ChapterConfigurationElement>()
            where (chapter.Title.Contains("and")) &&
             ((int.Parse(chapter.Number) % 2) == 0)
            select new
            {
                ChapterNumber = $"Chapter {chapter.Number}",
                chapter.Title
            };
```

```
foreach (var chapterInfo in expr)
    Console.WriteLine($"{chapterInfo.ChapterNumber} : {chapterInfo.Title}");
```

要查询的配置节如下所示。

```
<CSharpRecipesConfiguration CurrentEdition="4">
  <Chapters>
    <add Number="1" Title="Classes and Generics" />
    <add Number="2" Title="Collections, Enumerators, and Iterators" />
    <add Number="3" Title="Data Types" />
    <add Number="4" Title="LINQ &amp; Lambda Expressions" />
    <add Number="5" Title="Debugging and Exception Handling" />
    <add Number="6" Title="Reflection and Dynamic Programming" />
    <add Number="7" Title="Regular Expressions" />
    <add Number="8" Title="Filesystem I/O" />
    <add Number="9" Title="Networking and Web" />
    <add Number="10" Title="XML" />
    <add Number="11" Title="Security" />
    <add Number="12" Title="Threading, Synchronization, and Concurrency" />
    <add Number="13" Title="Toolbox" />
  </Chapters>
  <Editions>
    <add Number="1" PublicationYear="2004" />
    <add Number="2" PublicationYear="2006" />
    <add Number="3" PublicationYear="2007" />
    <add Number="4" PublicationYear="2015" />
  </Editions>
</CSharpRecipesConfiguration>
```

查询的输出如下所示。

```
Chapter 2 : Collections, Enumerators, and Iterators
Chapter 6 : Reflection and Dynamic Programming
Chapter 12 : Threading, Synchronization, and Concurrency
```

4.7.3 讨论

.NET 中的配置文件在实现基于 .NET 的应用程序的可管理性和易于部署方面起着重要的作用。在配置文件的层次结构中获取可能影响应用程序的各种设置可能是一项具有挑战性的任务，因此在开发、测试、部署和管理应用程序期间，理解如何编写工具程序从而以编程方式检查配置文件设置就是一项非常有意义的工作。

> 为了访问配置类型，需要引用 `System.Configuration` 程序集。

即使 `ConfigurationElementCollection` 类（配置文件中数据集的基类）只支持 `IEnumerable` 而不支持 `IEnumerable<T>`，我们仍然可以利用它通过在集合上使用 `OfType<ChapterConfigurationElement>` 方法来获取所需的元素，这将从集合中选择此类型的元素。

```
var expr = from ChapterConfigurationElement chapter in
    recipeConfig.Chapters.OfType<ChapterConfigurationElement>()
```

ChapterConfigurationElement 是自定义的配置节类，它保存有章号和标题。

```
/// <summary>
/// 保存了配置文件中的章节信息
/// </summary>
public class ChapterConfigurationElement : ConfigurationElement
{
    /// <summary>
    /// 默认构造函数
    /// </summary>
    public ChapterConfigurationElement()
    {
    }

    /// <summary>
    /// 章节的编号
    /// </summary>
    [ConfigurationProperty("Number", IsRequired=true)]
    public string Number
    {
        get { return (string)this["Number"]; }
        set { this["Number"] = value; }
    }

    /// <summary>
    /// 章节的标题
    /// </summary>
    [ConfigurationProperty("Title", IsRequired=true)]
    public string Title
    {
        get { return (string)this["Title"]; }
        set { this["Title"] = value; }
    }
}
```

这一技术也可以用在像 machine.config 这样的标准配置文件上。这个示例确定 machine.config 中的哪些区域需要访问权限。对于这个集合，将使用 OfType<ConfigurationSection>，因为这是一个标准区域。

```
System.Configuration.Configuration machineConfig =
    ConfigurationManager.OpenMachineConfiguration();

var query = from ConfigurationSection section in
machineConfig.Sections.OfType<ConfigurationSection>()
            where section.SectionInformation.RequirePermission
            select section;

foreach (ConfigurationSection section in query)
    Console.WriteLine(section.SectionInformation.Name);
```

检测到的节看起来如下所示。

```
configProtectedData
satelliteassemblies
assemblyBinding
system.codedom
system.data.dataset
system.data.odbc
system.data
system.data.oracleclient
system.data.oledb
uri
system.windows.forms
system.runtime.remoting
runtime
system.diagnostics
windows
mscorlib
system.webServer
system.data.sqlclient
startup
```

4.7.4 参考

MSDN 文档中的“Enumerable.ofType 方法”“ConfigurationSectionCollection 类”和“ConfigurationElementCollection 类”主题。

4.8 从数据库直接创建XML文件

4.8.1 问题

你希望能够从数据库中获取一个数据集并将其表示为 XML 文件。

4.8.2 解决方案

使用 LINQ to SQL 和 LINQ to XML 在一个查询中检索和转换数据。在这个示例中，我们选择 Northwind 数据库中的前 5 名客户，他们的联系方式是所有者，并且这些所有者所下订单的总额都超过 10 000 美元，然后创建一个 XML 文件，其中包含公司名称、联系人姓名、电话号码和订单总额。最后，将结果写入 BigSpenders.xml 文件。

```
NorthwindEntities dataContext = new NorthwindEntities();
// 将生成的SQL语句记录到控制台
dataContext.Database.Log = Console.WriteLine;

// 选择前5名客户,他们的联系方式是所有者
// 并且这些所有者所下订单的总额都超过10 000美元
var bigSpenders = new XElement("BigSpenders",
    from top5 in
        (
            (from customer in
                    (
                        from c in dataContext.Customers
```

```
                        // 获得联系方式是所有者
                        // 并且下过订单的客户
                    where c.ContactTitle.Contains("Owner")
                    && c.Orders.Count > 0
                    join orderData in
                        (
                             from c in dataContext.Customers
                             // 获得联系方式是所有者
                             // 并且下过订单的客户
                             where c.ContactTitle.Contains("Owner")
                             && c.Orders.Count > 0
                             from o in c.Orders
                                 // 获得订单明细
                             join od in dataContext.Order_Details
                                 on o.OrderID equals od.OrderID
                             select new
                             {
                                 c.CompanyName,
                                 c.CustomerID,
                                 o.OrderID,
                                 // 需要从订单明细中计算订单数值
                                 //(UnitPrice*Quantity as Total)-
                                 // (Total*Discount) as NetOrderTotal
                                 NetOrderTotal = (
                                     (((double)od.UnitPrice) * od.Quantity) -
                                     ((((double)od.UnitPrice) * od.Quantity) *
                                       od.Discount))
                             }
                        )
                    on c.CustomerID equals orderData.CustomerID
                    into customerOrders
                    select new
                    {
                        c.CompanyName,
                        c.ContactName,
                        c.Phone,
                        // 获得客户花费的总额
                        TotalSpend = customerOrders.Sum(order =>
                                        order.NetOrderTotal)
                    }
                )
                // 只考虑花费大于10 000的客户
            where customer.TotalSpend > 10000
            orderby customer.TotalSpend descending
            // 仅取花费前5名
            select new
            {
                CompanyName = customer.CompanyName,
                ContactName = customer.ContactName,
                Phone = customer.Phone,
                TotalSpend = customer.TotalSpend
            }).Take(5)
    ).ToList()
// 将数据格式化为XML
select new XElement("Customer",
```

```
            new XAttribute("companyName", top5.CompanyName),
            new XAttribute("contactName", top5.ContactName),
            new XAttribute("phoneNumber", top5.Phone),
            new XAttribute("amountSpent", top5.TotalSpend)));
using (XmlWriter writer = XmlWriter.Create("BigSpenders.xml"))
{
    bigSpenders.WriteTo(writer);
}
```

在构建更大的查询时，如果你更擅长 C# 而不是 SQL，那么可能会发现使用函数方式（`.Join()`）代替采用查询表达式方法（`join x on y equals z`）来构建查询更容易一些。

4.8.3 讨论

LINQ to SQL 是 LINQ to ADO.NET 的一部分，便于快速进行数据库开发。它适用于几乎直接对数据库模式进行编程的情况，在其中的大多数情况下，强类型化的类与数据库表之间具有一对一相关性。如果你处于一种更加偏向企业开发的情况，其中有许多存储过程和数据库已经不属于“一个表等于一个实体的情况”，那么最好调查一下 LINQ to Entity。

要访问 LINQ to SQL 可视化设计器，可以通过向项目中添加一个新的“LINQ to SQL 类”项或者打开一个现有的“LINQ to SQL 类”项（*.dbml 文件）打开此设计器。这将帮助你为数据库构建 `DataContext` 和实体类，然后可以把它们用于 LINQ（或者其他的编程构造，如果你希望这样做的话）。`DataContext` 类似于融为一体的 ADO.NET `Connection` 和 `Command` 对象。你使用它来建立连接、执行查询或者通过实体类直接访问表。`NorthwindLinq2Sql` 数据上下文是 `DataContext` 的一个强类型化的实例，其中部分内容如下所示。

```
public partial class NorthwindLinq2SqlDataContext : System.Data.Linq.DataContext
{
    private static System.Data.Linq.Mapping.MappingSource mappingSource = new
AttributeMappingSource();

#region Extensibility Method Definitions
partial void OnCreated();
partial void InsertCategory(Category instance);
partial void UpdateCategory(Category instance);
partial void DeleteCategory(Category instance);
partial void InsertTerritory(Territory instance);
partial void UpdateTerritory(Territory instance);
partial void DeleteTerritory(Territory instance);
partial void InsertCustomerCustomerDemo(CustomerCustomerDemo instance);
partial void UpdateCustomerCustomerDemo(CustomerCustomerDemo instance);
partial void DeleteCustomerCustomerDemo(CustomerCustomerDemo instance);
partial void InsertCustomerDemographic(CustomerDemographic instance);
partial void UpdateCustomerDemographic(CustomerDemographic instance);
partial void DeleteCustomerDemographic(CustomerDemographic instance);
partial void InsertCustomer(Customer instance);
partial void UpdateCustomer(Customer instance);
partial void DeleteCustomer(Customer instance);
partial void InsertEmployee(Employee instance);
```

```
partial void UpdateEmployee(Employee instance);
partial void DeleteEmployee(Employee instance);
partial void InsertEmployeeTerritory(EmployeeTerritory instance);
partial void UpdateEmployeeTerritory(EmployeeTerritory instance);
partial void DeleteEmployeeTerritory(EmployeeTerritory instance);
partial void InsertOrder_Detail(Order_Detail instance);
partial void UpdateOrder_Detail(Order_Detail instance);
partial void DeleteOrder_Detail(Order_Detail instance);
partial void InsertOrder(Order instance);
partial void UpdateOrder(Order instance);
partial void DeleteOrder(Order instance);
partial void InsertProduct(Product instance);
partial void UpdateProduct(Product instance);
partial void DeleteProduct(Product instance);
partial void InsertRegion(Region instance);
partial void UpdateRegion(Region instance);
partial void DeleteRegion(Region instance);
partial void InsertShipper(Shipper instance);
partial void UpdateShipper(Shipper instance);
partial void DeleteShipper(Shipper instance);
partial void InsertSupplier(Supplier instance);
partial void UpdateSupplier(Supplier instance);
partial void DeleteSupplier(Supplier instance);
#endregion

    public NorthwindLinq2SqlDataContext() :
     base(
 global::NorthwindLinq2Sql.Properties.Settings.Default.NorthwindConnectionString,
     mappingSource)    {
        OnCreated();
    }

    public NorthwindLinq2SqlDataContext(string connection) :
            base(connection, mappingSource)
    {
        OnCreated();
    }

    public NorthwindLinq2SqlDataContext(System.Data.IDbConnection connection) :
            base(connection, mappingSource)
    {
        OnCreated();
    }

    public NorthwindLinq2SqlDataContext(string connection,
        System.Data.Linq.Mapping.MappingSource mappingSource) :
            base(connection, mappingSource)
    {
        OnCreated();
    }

    public NorthwindLinq2SqlDataContext(System.Data.IDbConnection connection,
System.Data.Linq.Mapping.MappingSource mappingSource) :
            base(connection, mappingSource)
    {
```

```
        OnCreated();
    }

    public System.Data.Linq.Table<Category> Categories
    {
        get
        {
            return this.GetTable<Category>();
        }
    }

    public System.Data.Linq.Table<Territory> Territories
    {
        get
        {
            return this.GetTable<Territory>();
        }
    }

    public System.Data.Linq.Table<CustomerCustomerDemo> CustomerCustomerDemos
    {
        get
        {
            return this.GetTable<CustomerCustomerDemo>();
        }
    }

    public System.Data.Linq.Table<CustomerDemographic> CustomerDemographics
    {
        get
        {
            return this.GetTable<CustomerDemographic>();
        }
    }

    public System.Data.Linq.Table<Customer> Customers
    {
        get
        {
            return this.GetTable<Customer>();
        }
    }

    public System.Data.Linq.Table<Employee> Employees
    {
        get
        {
            return this.GetTable<Employee>();
        }
    }

    public System.Data.Linq.Table<EmployeeTerritory> EmployeeTerritories
    {
        get
        {
```

```
                return this.GetTable<EmployeeTerritory>();
            }
        }

        public System.Data.Linq.Table<Order_Detail> Order_Details
        {
            get
            {
                return this.GetTable<Order_Detail>();
            }
        }

        public System.Data.Linq.Table<Order> Orders
        {
            get
            {
                return this.GetTable<Order>();
            }
        }

        public System.Data.Linq.Table<Product> Products
        {
            get
            {
                return this.GetTable<Product>();
            }
        }

        public System.Data.Linq.Table<Region> Regions
        {
            get
            {
                return this.GetTable<Region>();
            }
        }

        public System.Data.Linq.Table<Shipper> Shippers
        {
            get
            {
                return this.GetTable<Shipper>();
            }
        }

        public System.Data.Linq.Table<Supplier> Suppliers
        {
            get
            {
                return this.GetTable<Supplier>();
            }
        }
    }
```

Northwind 数据库的实体类也都展示在生成的代码中，其中每个表都定义了一个实体类。实体类通过不带参数的 `Table` 特性指示。这意味着实体类的名称与表的名称相匹配。

```
[global::System.Data.Linq.Mapping.TableAttribute(Name="dbo.Customers")]
public partial class Customer : INotifyPropertyChanging, INotifyPropertyChanged
{
    private static PropertyChangingEventArgs emptyChangingEventArgs = new
PropertyChangingEventArgs(String.Empty);

    private string _CustomerID;

    private string _CompanyName;

    private string _ContactName;

    private string _ContactTitle;

    private string _Address;

    private string _City;

    private string _Region;

    private string _PostalCode;

    private string _Country;

    private string _Phone;

    private string _Fax;

    private EntitySet<CustomerCustomerDemo> _CustomerCustomerDemos;

    private EntitySet<Order> _Orders;

#region Extensibility Method Definitions
partial void OnLoaded();
partial void OnValidate(System.Data.Linq.ChangeAction action);
partial void OnCreated();
partial void OnCustomerIDChanging(string value);
partial void OnCustomerIDChanged();
partial void OnCompanyNameChanging(string value);
partial void OnCompanyNameChanged();
partial void OnContactNameChanging(string value);
partial void OnContactNameChanged();
partial void OnContactTitleChanging(string value);
partial void OnContactTitleChanged();
partial void OnAddressChanging(string value);
partial void OnAddressChanged();
partial void OnCityChanging(string value);
partial void OnCityChanged();
partial void OnRegionChanging(string value);
partial void OnRegionChanged();
partial void OnPostalCodeChanging(string value);
partial void OnPostalCodeChanged();
partial void OnCountryChanging(string value);
partial void OnCountryChanged();
partial void OnPhoneChanging(string value);
```

```
    partial void OnPhoneChanged();
    partial void OnFaxChanging(string value);
    partial void OnFaxChanged();
    #endregion

        public Customer()
        {
            this._CustomerCustomerDemos = new EntitySet<CustomerCustomerDemo>(new
    Action<CustomerCustomerDemo>(this.attach_CustomerCustomerDemos), new
    Action<CustomerCustomerDemo>(this.detach_CustomerCustomerDemos));
            this._Orders = new EntitySet<Order>(
                new Action<Order>(this.attach_Orders), new
    Action<Order>(this.detach_Orders));
            OnCreated();
        }

        public event PropertyChangingEventHandler PropertyChanging;

        public event PropertyChangedEventHandler PropertyChanged;

        protected virtual void SendPropertyChanging()
        {
            if ((this.PropertyChanging != null))
            {
                this.PropertyChanging(this, emptyChangingEventArgs);
            }
        }

        protected virtual void SendPropertyChanged(String propertyName)
        {
            if ((this.PropertyChanged != null))
            {
                this.PropertyChanged(this,
                    new PropertyChangedEventArgs(propertyName));
            }
        }
```

标准属性改变通知是通过 INotifyPropertyChanging 和 INotifyPropertyChanged 实现的，并且包含 PropertyChanging 和 PropertyChanged 事件，用于传达属性的改变。还有一组分部方法，如果是在实体类的另一个分部类定义中实现了分部方法，那么在这个实体类上修改属性时，它将会调用这些方法。

> 很多 Microsoft .NET 生成的类都是作为分部类生成的。这样就可以在你自己的分部类中扩展它们，并向类添加方法和属性，而不会在代码生成器下次重新生新代码时产生混乱的代码。
>
> 在这种情况下，如果没有发现其他的分部类定义，编译器将删除这些通知。分部方法允许在分部类声明的一个文件中声明方法签名，并且在另一个文件中实现该方法。如果编译器找到了签名但没有找到实现，它就会删除该签名。

实体类中的属性通过 Column 特性与数据库中的列匹配，其中 Name 值是数据库列名，Storage 值是类数据的内部存储。用于改变属性的事件被编码进属性的设置器中。

```
[global::System.Data.Linq.Mapping.ColumnAttribute(Storage="_CompanyName",
DbType="NVarChar(40) NOT NULL", CanBeNull=false)]
public string CompanyName
{
    get
    {
        return this._CompanyName;
    }
    set
    {
        if ((this._CompanyName != value))
        {
            this.OnCompanyNameChanging(value);
            this.SendPropertyChanging();
            this._CompanyName = value;
            this.SendPropertyChanged("CompanyName");
            this.OnCompanyNameChanged();
        }
    }
}
```

对于一对多子关系，可以利用 Association 特性声明子实体类的 EntitySet<T>。Association 特性指定了父实体类与子实体类之间的关系信息，如下面用于 Customer 上的 Orders 属性的代码所示。

```
[global::System.Data.Linq.Mapping.AssociationAttribute(Name="Customer_Order",
Storage="_Orders", ThisKey="CustomerID", OtherKey="CustomerID")]
public EntitySet<Order> Orders
{
    get
    {
        return this._Orders;
    }
    set
    {
        this._Orders.Assign(value);
    }
}
```

LINQ to SQL 能够做的事情远不止上面展示的这些，我们鼓励你更深入地调查它。但是现在让我们转换到将要处理的另一个数据域：LINQ to XML。

LINQ to XML 不仅涉及如何对 XML 执行查询，而且是一种对开发人员更友好的 XML 处理方式。LINQ to XML 中的主要类之一是 XElement，它允许你以一种更类似于 XML 自身结构的方式创建 XML。这似乎并不是很重要，但是当你可以在代码中看到 XML 的形态时，就更容易知道你所处的位置。（曾经忘记过处在哪个 XmlWriter.WriteEndElment 上吗？我们就有过这种经历！）在第 10 章中可以获得关于使用 XElement 的详细信息和示例，因此我们不会在此处作进一步的介绍，但是如你所见，在查询中构建 XML 是很容易的。

查询的第一部分涉及建立主要的 XML 元素 BigSpenders，以及获取其联系方式是所有者的初始客户集合。

```
var bigSpenders = new XElement("BigSpenders",
            from top5 in
            (
                (from customer in
                     (
                         from c in dataContext.Customers
                         // 获取联系方式是所有者,
                         // 并且下过订单的客户
                         where c.ContactTitle.Contains("Owner")
                            && c.Orders.Count > 0
```

查询的中间部分涉及把订单和订单详细信息与客户信息联接起来，用以获取订单的 NetOrderTotal。它还会创建包含此值的订单数据、客户和订单 ID 以及公司名称。在查询的最后一部分中需要 NetOrderTotal，敬请期待！

```
        join orderData in
            (
                from c in dataContext.Customers
                // 获得联系信息是所有者,
                // 并且下过订单的客户
                where c.ContactTitle.Contains("Owner")
                   && c.Orders.Count > 0
                from o in c.Orders
                // 获得订单明细
                join od in dataContext.OrderDetails
                    on o.OrderID equals od.OrderID
                select new
                {
                    c.CompanyName,
                    c.CustomerID,
                    o.OrderID,
                    // 需要从订单明细中计算订单数值
                    //(UnitPrice*Quantity as Total)
                    //(Total*Discount)
                    // as NetOrderTotal
                    NetOrderTotal = (
    (((double)od.UnitPrice) * od.Quantity) -
    ((((double)od.UnitPrice) * od.Quantity) * od.Discount))
                }
            )
        on c.CustomerID equals orderData.CustomerID
        into customerOrders
```

查询的最后一部分对生成的 customerOrders 集合中的 NetOrderTotal 应用 Sum 函数，跨所有订单确定该客户的 TotalSpend。最后，查询使用 Take 函数只选择其 TotalSpend 值大于 10 000 的前 5 名客户。（Take 等价于 SQL 中的 TOP。）然后使用这些记录构造一个内部 Customer 元素，其属性嵌套开始于查询第一部分中的 BigSpenders 根元素内。

```
                select new
                {
                     c.CompanyName,
                     c.ContactName,
                     c.Phone,
                     // 获得客户花费的总额
```

```
                    TotalSpend = customerOrders.Sum(order => order. NetOrderTotal)
                }
             )
        // 只考虑花费大于10 000的客户
        where customer.TotalSpend > 10000
        orderby customer.TotalSpend descending
       // 仅取花费前5名
       select customer).Take(5)
)
// 将数据格式化为XML
select new XElement("Customer",
           new XAttribute("companyName", top5.CompanyName),
           new XAttribute("contactName", top5.ContactName),
           new XAttribute("phoneNumber", top5.Phone),
           new XAttribute("amountSpent", top5.TotalSpend)));
```

> 一种更容易的方法是，首先把大型嵌套查询构建为单独的查询，一旦确信内部查询生效，就把它们组合在一起。

此时，对于这里的所有代码，还没有发生任何事情。这是对的：在访问查询之前，因为延迟执行而导致什么也没有发生。LINQ 构造了一个查询表达式，但是没有对数据库做什么事情；内存中也没有 XML。一旦在 `bigSpenders` 查询表达式上调用 `WriteTo` 方法，LINQ to SQL 就会对查询求值，并构造 XML。`WriteTo` 方法将把构造的 XML 写入提供的 `XmlWriter`，我们就完成了整个操作。

```
using (XmlWriter writer = XmlWriter.Create("BigSpenders.xml"))
{
    bigSpenders.WriteTo(writer);
}
```

如果你对 SQL 代码感兴趣，可以把 `DataContext.Log` 属性连接到一个 `TextWriter`（例如控制台）。

```
// 将生成的SQL语句记录到控制台
dataContext.Log = Console.Out;
```

这个查询生成的 SQL 代码如下所示。

```
    Generated SQL for query - output via DataContext.Log
    SELECT [t10].[CompanyName], [t10].[ContactName], [t10].[Phone],
        [t10].[TotalSpend]
    FROM (
        SELECT TOP (5) [t0].[Company Name] AS [CompanyName],
            [t0].[Contact Name] AS
    [ContactName], [t0].[Phone], [t9].[value] AS [TotalSpend]
        FROM [Customers] AS [t0]
        OUTER APPLY (
            SELECT COUNT(*) AS [value]
            FROM [Orders] AS [t1]
            WHERE [t1].[Customer ID] = [t0].[Customer ID]
```

```
            ) AS [t2]
        OUTER APPLY (
            SELECT SUM([t8].[value]) AS [value]
            FROM (
                SELECT [t3].[Customer ID], [t6].[Order ID],
                     ([t7].[Unit Price] *
                     (CONVERT(Decimal(29,4),[t7].[Quantity]))) -
                         ([t7].[Unit Price] *
                         (CONVERT(Decimal(29,4),[t7].[Quantity])) *
                             (CONVERT(Decimal(29,4),[t7].[Discount]))) AS
                                 [value],
                     [t7].[Order ID] AS [Order ID2],
                     [t3].[Contact Title] AS [ContactTitle],
                     [t5].[value] AS [value2],
                     [t6].[Customer ID] AS [CustomerID]
            FROM [Customers] AS [t3]
            OUTER APPLY (
                SELECT COUNT(*) AS [value]
                FROM [Orders] AS [t4]
                WHERE [t4].[Customer ID] = [t3].[Customer ID]
                ) AS [t5]
            CROSS JOIN [Orders] AS [t6]
            CROSS JOIN [Order Details] AS [t7]
            ) AS [t8]
        WHERE ([t0].[Customer ID] = [t8].[Customer ID]) AND ([t8].[Order ID] = [
t8].[Order ID2]) AND ([t8].[ContactTitle] LIKE @p0) AND ([t8].[value2] > @p1) AN
D ([t8].[CustomerID] = [t8].[Customer ID])
            ) AS [t9]
        WHERE ([t9].[value] > @p2) AND ([t0].[Contact Title] LIKE @p3) AND
            ([t2].[va
    lue] > @p4)
        ORDER BY [t9].[value] DESC
        ) AS [t10]
    ORDER BY [t10].[TotalSpend] DESC
    -- @p0: Input String (Size = 0; Prec = 0; Scale = 0) [%Owner%]
    -- @p1: Input Int32 (Size = 0; Prec = 0; Scale = 0) [0]
    -- @p2: Input Decimal (Size = 0; Prec = 29; Scale = 4) [10000]
    -- @p3: Input String (Size = 0; Prec = 0; Scale = 0) [%Owner%]
    -- @p4: Input Int32 (Size = 0; Prec = 0; Scale = 0) [0]
    -- Context: SqlProvider(SqlCE) Model: AttributedMetaModel Build: 3.5.20706.1
```

最终的 XML 如下所示。

```
<BigSpenders>
  <Customer companyName="Folk och fa HB" contactName="Maria Larsson"
            phoneNumber="0695-34 67 21" amountSpent="39805.162472039461" />
  <Customer companyName="White Clover Markets" contactName="Karl Jablonski"
            phoneNumber="(206) 555-4112" amountSpent="35957.604972146451" />
  <Customer companyName="Bon app'" contactName="Laurence Lebihan"
            phoneNumber="91.24.45.40" amountSpent="22311.577472746558" />
  <Customer companyName="LINO-Delicateses" contactName="Felipe Izquierdo"
            phoneNumber="(8) 34-56-12" amountSpent="20458.544984650609" />
  <Customer companyName="Simons bistro" contactName="Jytte Petersen"
            phoneNumber="31 12 34 56" amountSpent="18978.777493602414" />
</BigSpenders>
```

4.8.4 参考

MSDN 文档中的“Introduction to LINQ Queries”“DataContext.Log 属性”“DataContext 类”“XElement 类”和“LINQ to SQL”主题。

4.9 有选择地输出查询结果

4.9.1 问题

你希望能够获取查询结果的动态子集。

4.9.2 解决方案

使用 TakeWhile 扩展方法获取与条件匹配的所有结果，直到遇到第一个不满足条件的记录。

```
NorthwindEntities dataContext = new NorthwindEntities();

// 找到所有供应商的产品
var query =
    dataContext.Suppliers.GroupJoin(dataContext.Products,
        s => s.SupplierID, p => p.SupplierID,
        (s, products) => new
        {
            s.CompanyName,
            s.ContactName,
            s.Phone,
            Products = products
        }).OrderByDescending(supplierData => supplierData.Products.Count());

var results =
    query.AsEnumerable().TakeWhile(supplierData =>
        supplierData.Products.Count() > 3);
Console.WriteLine($"Suppliers that provide more than three products: " +
    $"{results.Count()}");
foreach (var supplierData in results)
{
    Console.WriteLine($"  Company Name : {supplierData.CompanyName}");
    Console.WriteLine($"  Contact Name : {supplierData.ContactName}");
    Console.WriteLine($"  Contact Phone : {supplierData.Phone}");
    Console.WriteLine($"  Products Supplied : {supplierData.Products.Count()}");
    foreach (var productData in supplierData.Products)
        Console.WriteLine($"        Product: {productData.ProductName}");
}
```

一旦遇到不满足条件的记录，你也可以使用 SkipWhile 扩展方法获取所有结果。

```
NorthwindEntities dataContext = new NorthwindEntities();

// 找到所有供应商的产品
var query =
    dataContext.Suppliers.GroupJoin(dataContext.Products,
```

```
            s => s.SupplierID, p => p.SupplierID,
            (s, products) => new
            {
                s.CompanyName,
                s.ContactName,
                s.Phone,
                Products = products
            }).OrderByDescending(supplierData => supplierData.Products.Count());

    var results =
        query.AsEnumerable().SkipWhile(supplierData =>
        supplierData.Products.Count() > 3);
    Console.WriteLine($"Suppliers that provide more than three products: " +
        $"{results.Count()}");
    foreach (var supplierData in results)
    {
        Console.WriteLine($"    Company Name : {supplierData.CompanyName}");
        Console.WriteLine($"    Contact Name : {supplierData.ContactName}");
        Console.WriteLine($"    Contact Phone : {supplierData.Phone}");
        Console.WriteLine($"    Products Supplied : {supplierData.Products.Count()}");
        foreach (var productData in supplierData.Products)
            Console.WriteLine($"        Product: {productData.ProductName}");
    }
```

4.9.3 讨论

在这个使用 LINQ to SQL 的示例中，确定每个供应商提供的产品数量，并按产品数以降序对结果排序。

```
    var query =
        dataContext.Suppliers.GroupJoin(dataContext.Products,
            s => s.SupplierID, p => p.SupplierID,
            (s, products) => new
            {
                s.CompanyName,
                s.ContactName,
                s.Phone,
                Products = products
            }).OrderByDescending(supplierData => supplierData.Products.Count());
```

从该结果中，仅当供应商提供了 3 件以上的产品时，才会把该供应商数据纳入最终结果集中，并显示结果。`TakeWhile` 与 lambda 表达式一起用于确定产品数是否大于 3；如果是，就把供应商纳入结果集中。

```
    var results = query.AsEnumerable().TakeWhile(supplierData =>
            supplierData.Products.Count() > 3);
```

如果代之以使用 `SkipWhile`，则将返回提供了 3 件或 3 件以下产品的所有供应商，代码如下所示。

```
    var results = query.AsEnumerable().SkipWhile(supplierData =>
            supplierData.Products.Count() > 3);
```

能够编写基于代码的条件提供了比常规的 Take 和 Skip 方法（它们基于绝对记录数）更大的灵活性，但要记住，一旦达到 TakeWhile 或 SkipWhile 的条件之后，你就获得了所有的记录。因此，在使用它们之前对结果集进行排序很重要。

查询还使用了 GroupJoin，它类似于 SQL 中的 LEFT OUTER JOIN 或 RIGHT OUTER JOIN，但不会平铺结果。GroupJoin 会产生层次式的结果集，而不是表格式的结果集。在这个示例中，它用于获得供应商的产品集合。

```
dataContext.Suppliers.GroupJoin(dataContext.Products,
   s => s.SupplierID, p => p.SupplierID,
```

以下是 TakeWhite 的输出。

```
Suppliers that provide more than three products: 4
    Company Name : Pavlova, Ltd.
    Contact Name : Ian Devling
    Contact Phone : (03) 444-2343
    Products Supplied : 5
        Product: Pavlova
        Product: Alice Mutton
        Product: Carnarvon Tigers
        Product: Vegie-spread
        Product: Outback Lager
    Company Name : Plutzer Lebensmittelgroßmärkte AG
    Contact Name : Martin Bein
    Contact Phone : (069) 992755
    Products Supplied : 5
        Product: Rössle Sauerkraut
        Product: Thüringer Rostbratwurst
        Product: Wimmers gute Semmelknödel
        Product: Rhönbräu Klosterbier
        Product: Original Frankfurter grüne Soße
    Company Name : New Orleans Cajun Delights
    Contact Name : Shelley Burke
    Contact Phone : (100) 555-4822
    Products Supplied : 4
        Product: Chef Anton's Cajun Seasoning
        Product: Chef Anton's Gumbo Mix
        Product: Louisiana Fiery Hot Pepper Sauce
        Product: Louisiana Hot Spiced Okra
    Company Name : Specialty Biscuits, Ltd.
    Contact Name : Peter Wilson
    Contact Phone : (161) 555-4448
    Products Supplied : 4
        Product: Teatime Chocolate Biscuits
        Product: Sir Rodney's Marmalade
        Product: Sir Rodney's Scones
        Product: Scottish Longbreads
```

4.9.4 参考

MSDN 文档中的“Enumerable.TakeWhile 方法”“Enumerable.SkipWhile 方法”和“Enumerable.GroupJoin 方法”主题。

4.10 将LINQ用于不支持IEnumerable<T>的集合

4.10.1 问题

有一大批集合不支持 `IEnumerable` 或 `ICollection` 的泛型版本，但是却支持 `IEnumerable` 或 `ICollection` 接口的原始非泛型版本，你希望能够使用 LINQ 查询这些集合。

4.10.2 解决方案

不能从原始 `IEnumerable` 或 `ICollection` 接口推断类型，因此必须使用 `OfType<T>` 或 `Cast<T>` 扩展方法提供它，或者在 `from` 子句中指定类型，它将为你插入一个 `Cast<T>`。第一个示例使用 `Cast<XmlNode>` 让 LINQ 知道从 `XmlDocument.SelectNodes` 返回的 `XmlNodeList` 中的元素是 `XmlNode` 类型。有关如何使用 `OfType<T>` 扩展方法的示例，参见 4.10.3 节。

```
// 创建一些XML以在LINQ中使用这些
// 不直接支持IEnumerable<T>的类型
XElement xmlFragment = new XElement("NonGenericLinqableTypes",
                        new XElement("IEnumerable",
                            new XElement("System.Collections",
                                new XElement("ArrayList"),
                                new XElement("BitArray"),
                                new XElement("Hashtable"),
                                new XElement("Queue"),
                                new XElement("SortedList"),
                                new XElement("Stack")),
                            new XElement("System.Net",
                                new XElement("CredentialCache")),
                            new XElement("System.Xml",
                                new XElement("XmlNodeList")),
                            new XElement("System.Xml.XPath",
                                new XElement("XPathNodeIterator"))),
                        new XElement("ICollection",
                            new XElement("System.Diagnostics",
                                new XElement("EventLogEntryCollection")),
                            new XElement("System.Net",
                                new XElement("CookieCollection")),
                            new XElement("System.Security.AccessControl",
                                new XElement("GenericAcl")),
                            new XElement("System.Security",
                                new XElement("PermissionSet"))));

XmlDocument doc = new XmlDocument();
doc.LoadXml(xmlFragment.ToString());

// 选择位于IEnumerable之下包含子元素，名称为System.Collections
// 并且名称中包含大写字母S的节点的名称，
// 以降序返回结果列表
var query =
from node in
    doc.SelectNodes("/NonGenericLinqableTypes/IEnumerable/*").Cast<XmlNode>()
    where node.HasChildNodes &&
```

```
            node.Name == "System.Collections"
        from XmlNode xmlNode in node.ChildNodes
        where xmlNode.Name.Contains('S')
        orderby xmlNode.Name descending
        select xmlNode.Name;

foreach (string name in query)
    Console.WriteLine(name);
```

第二个示例操作应用程序事件日志，并检索在过去 6 小时内发生的错误。在 from 关键字后面提供了集合中的元素类型（EventLogEntry），它允许 LINQ 推断它所需的关于集合元素类型的其余信息。

```
EventLog log = new EventLog("Application");
query = from EventLogEntry entry in log.Entries
        where entry.EntryType == EventLogEntryType.Error &&
            entry.TimeGenerated > DateTime.Now.Subtract(new TimeSpan(6, 0, 0))
        select entry.Message;

Console.WriteLine($"There were {query.Count<string>()}" +
    " Application Event Log error messages in the last 6 hours!");
foreach (string message in query)
    Console.WriteLine(message);
```

4.10.3　讨论

Cast<T> 会将 IEnumerable 转换成 IEnumerable<T>，使得 LINQ 可以以一种强类型化的方式访问集合中的每一项。在使用 Cast<T> 之前，有必要检查集合中的所有元素确实都是类型 T，否则如果元素的类型不能转换成指定的类型 T，会得到一个 InvalidCastException，因为将使用类型强制转换所有的元素。把元素的类型置于 from 关键字后面时，其作用就像 Cast<T> 一样。

```
ArrayList stuff = new ArrayList();
stuff.Add(DateTime.Now);
stuff.Add(DateTime.Now);
stuff.Add(1);
stuff.Add(DateTime.Now);

var expr = from item in stuff.Cast<DateTime>()
            select item;
foreach (DateTime item in expr)
    Console.WriteLine(item);
```

> 由于延迟执行语义，仅当迭代到此元素时，才会触发有关 Cast<T> 或 from 的异常。

解决这个问题的另一种方式是使用 OfType<T>，因为它只会返回特定类型的元素，而不会尝试把元素从一种类型强制转换为另一种类型。

```
var expr = from item in stuff.OfType<DateTime>()
           select item;
// 仅有三个都是DateTime的元素返回
// 不会触发异常
foreach (DateTime item in expr)
    Console.WriteLine(item);
```

4.10.4　参考

MSDN 文档中的“OfType<TResult> 方法”和“Cast<TResult> 方法”主题。

4.11　执行高级接口查找

4.11.1　问题

你将使用 Type 类查找某个接口。不过，Type 对象的 GetInterface 和 GetInterfaces 方法没有提供复杂的接口查找。

4.11.2　解决方案

使用 LINQ 查询类型接口信息，并执行丰富的查找条件。例 4-2 中所示的方法将演示 LINQ 可以执行的一种复杂的查找。

例 4-2：对类型上的接口执行复杂的查找

```
// 设置要查找的接口
Type[] interfaces = {
    typeof(System.ICloneable),
    typeof(System.Collections.ICollection),
    typeof(System.IAppDomainSetup) };

// 设置要检查的类型
Type searchType = typeof(System.Collections.ArrayList);

var matches = from t in searchType.GetInterfaces()
              join s in interfaces on t equals s
              select s;

Console.WriteLine("Matches found:");
foreach (Type match in matches)
    Console.WriteLine(match.ToString());
```

例 4-2 中的代码搜索 interfaces 数组中包含的三个接口中被 System.Collections.ArrayList 类型实现的任意接口。这是通过使用 LINQ 来连接类型实现的接口集合及 interfaces 数组来实现的。

GetInterface 方法只按名称查找接口（使用区分大小写或不区分大小写的查找），GetInterfaces 方法则返回特定类型上实现的所有接口的数组。要执行一个更特定的查询 [例如，找定义具有特定签名的方法的接口，或者实现从全局程序集缓存（GAC，其中存储公共程序集）加载的接口]，你需要使用一种不同的机制，例如 LINQ。LINQ 为查找接

口提供了一种更灵活、更高级的能力，而不需要创建自己的接口查找引擎。这种能力可以用于加载特定接口的程序集，从现有程序集中生成代码甚至用作一种逆向工程工具。

4.11.3 讨论

使用 LINQ 来查找在某种类型上实现的接口可以有许多种方式。下面只列出了可以执行的其他少数几种查找方式。

- 查找在特定命名空间（在本例中是 System.Collections 命名空间）内定义的所有实现的接口。

```
var collectionsInterfaces = from type in searchType.GetInterfaces()
                            where type.Namespace == "System.Collections"
                            select type;
```

- 查找包含返回一个 Int32 值 Add 方法的所有实现的接口。

```
var addInterfaces = from type in searchType.GetInterfaces()
                    from method in type.GetMethods()
                    where (method.Name == "Add") &&
                            (method.ReturnType == typeof(int))
                    select type;
```

- 查找从 GAC 加载的所有实现的接口。

```
var gacInterfaces = from type in searchType.GetInterfaces()
                    where type.Assembly.GlobalAssemblyCache
                    select type;
```

- 查找版本号为 4.0.0.0 的程序集内定义的所有实现的接口。

```
var versionInterfaces = from type in searchType.GetInterfaces()
                        where type.Assembly.GlobalAssemblyCache &&
                            type.Assembly.GetName().Version.Major == 4 &&
                            type.Assembly.GetName().Version.Minor == 0 &&
                            type.Assembly.GetName().Version.Build == 0 &&
                            type.Assembly.GetName().Version.Revision == 0
                        select type;
```

4.11.4 参考

MSDN 文档中的“lambda 表达式（C# 编程指南）”和“where 关键字［LINQ］（C#）”主题。

4.12 使用lambda表达式

4.12.1 问题

C# 中包含一个称为 lambda 表达式的特性。虽然可以将 lambda 表达式视为降低定义匿名方法难度的语法糖，但是也要理解它们在日常编程任务中的所有不同应用以及这些应用带来的结果。

4.12.2　解决方案

编译器可以从开发人员创建的方法中实现 lambda 表达式。lambda 表达式可能具有以下两个正交特征。

- 参数列表可以是显式或隐式类型。
- 表达式体可以是表达式或语句块。

让我们从使用委托的普通方式开始。首先，将声明一种委托类型（此处是 DoWork），然后将创建它的一个实例（如 WorkItOut 方法中所示）。声明委托的实例要求指定当调用委托时要执行的方法，这里已经连接了 DoWorkMethodImpl 方法。调用委托，并通过 DoWorkMethodImpl 方法把文本写到控制台。

```
class OldWay
{
    // 声明委托
    delegate int DoWork(string work);

    // 编写一个方法以创建委托实例并调用委托
    public void WorkItOut()
    {
        // 声明实例
        DoWork dw = new DoWork(DoWorkMethodImpl);
        // 调用委托
        int i = dw("Do work the old way");
    }

    // 编写一个委托将连接的方法,具有相同的签名
    // 因此当委托被调用时,此方法被调用
    public int DoWorkMethodImpl(string s)
    {
        Console.WriteLine(s);
        return s.GetHashCode();
    }
}
```

lambda 表达式允许建立在调用委托时要运行的代码，但是不必给委托提供一个命名的正式方法声明。声明的方法是无名的，并且在外层方法的作用域上关闭。例如，可以使用 lambda 表达式编写前面的代码。

```
class LambdaWay
{
    // 声明委托
    delegate int DoWork(string work);

    // 编写一个方法以创建委托实例并调用委托
    public void WorkItOut()
    {
        // 声明实例
        DoWork dw = s =>
        {
            Console.WriteLine(s);
            return s.GetHashCode();
```

```
        };
        // 调用委托
        int i = dw("Do some inline work");
    }
}
```

我们注意到，这段代码中没有使用名为 `DoWorkMethodImpl` 的方法，而是使用 `=>` 运算符从内联到 `DoWork` 委托的那个方法指派代码，这种指派方法如下所示。

```
DoWork dw = s =>
{
    Console.WriteLine(s);
    return s.GetHashCode();
};
```

你还提供了 `DoWork` 委托需要的参数（`string`），并且代码根据委托的需要返回一个 `int`（`s.GetHashCode()`）。在建立 lambda 表达式时，代码必须与委托签名匹配，否则将得到一个编译器错误。

“匹配”有以下几个含义。

- 如果显式类型化，lambda 参数必须与委托参数精确匹配。如果隐式类型化，lambda 参数就会获得委托参数类型。
- lambda 体必须是被赋予参数类型的合法表达式或语句块。
- lambda 的返回类型必须可以隐式转换成委托的返回类型。它不需要精确匹配。

还可以用另一种方式建立委托，即通过委托推断来完成。委托推断允许把方法名直接赋予委托实例，而不必编写用于创建新委托对象的代码。在底层，C# 实际上会编写用于创建委托对象的 IL，但是此处不需要显式地这样做。使用委托推断而不是到处书写 `new[DelegateType]([MethodName])` 有助于使在使用委托时涉及的代码保持整洁，如下所示。

```
class DirectAssignmentWay
{
    // 声明委托
    delegate int DoWork(string work);

    // 编写一个方法以创建委托实例并调用委托
    public void WorkItOut()
    {
        // 声明实例并指派方法
        DoWork dw = DoWorkMethodImpl;
        // 调用委托
        int i = dw("Do some direct assignment work");
    }
    // 编写一个委托将连接的方法,具有相同的签名
    // 因此当委托被调用时,此方法被调用
    public int DoWorkMethodImpl(string s)
    {
        Console.WriteLine(s);
        return s.GetHashCode();
    }
}
```

我们注意到，赋予 DoWork 委托实例 dw 的全部内容只有方法名 DoWorkMethodImpl，不像旧的 C# 代码那样还有 new DoWork(DoWorkMethodImpl) 调用。

> 记住，底层委托包装器并没有消失；委托推断只是通过隐藏它的一些内容简化一下语法。

此外，还可以建立接受泛型类型参数的 lambda 表达式，用于泛型委托，如同此处的 GenericWay 类中所示。

```
class GenericWay
{
    // 编写一个方法以创建委托实例并调用委托
    public void WorkItOut()
    {
        Func<string, string> dwString = s =>
        {
            Console.WriteLine(s);
            return s;
        };

        // 调用string委托
        string retStr = dwString("Do some generic work");

        Func<int, int> dwInt = i =>
        {
            Console.WriteLine(i);
            return i;
        };

        // 调用int委托
        int j = dwInt(5);

    }
}
```

4.12.3　讨论

lambda 表达式的一个有用之处是外层变量的概念。外层变量的官方定义是，任何具有包含 lambda 表达式的作用域的局部变量、值参数或参数数组。

这意味着，在 lambda 表达式的代码内，可以接触到该方法的作用域外面的变量。有一个“捕获”变量的概念，捕获发生在 lambda 表达式实际引用外层变量之一时。在下面的示例中，lambda 表达式将捕获 count 变量并递增它。count 变量不是 lambda 表达式的一部分，而是外层作用域的一部分。它被递增，然后返回递增后的值并求和。

```
public void SeeOuterWork()
{
    int count = 0;
```

```
    int total = 0;
    Func<int> countUp = () => count++;
    for (int i = 0; i < 10; i++)
        total += countUp();
    Debug.WriteLine($"Total = {total}");
}
```

捕获操作实际的作用是延长外层变量的生存期，使之与代表 lambda 表达式的底层委托实例的生存期保持一致。这应该促使你小心使用从 lambda 表达式内接触到的内容。它可能会导致一些对象生存的时间比最初计划的要长许多。在 lambda 表达式中使用外层变量时，垃圾收集器在以后才会有机会清理这些变量。捕获外层变量对垃圾收集器有另一种作用，当捕获局部变量或值参数时，不再把它们视为固定的，而认为它们是活动的。因此在使用这些变量之前，不安全的代码必须使用 `fixed` 关键字固定它们。

外层变量可以影响编译器为 lambda 表达式生成内部 IL 的方式。如果 lambda 表达式使用了外层变量，就把该 lambda 表达式生成为嵌套类的一个私有方法。如果 lambda 表达式没有使用外层变量，则将其生成为声明它的类中的另一个私有方法。如果外层方法是静态的，那么 lambda 表达式就不能通过 `this` 关键字访问实例成员，因为嵌套类也会被生成为静态的。

有两类 lambda 表达式：表达式（expression）lambda 和语句（statement）lambda。表达式 lambda 不带参数，只简单地递增表达式中的 `count` 变量。

```
int count = 0;
Func<int> countUp = () => count++;
```

语句 lambda 具有封闭在花括号中的主体，可以包含任意数量的语句。

```
Func<int, int> dwInt = i =>
{
    Console.WriteLine(i);
    return i;
};
```

关于 lambda 表达式要记住的最后几点包括以下这些。

- 它们不能使用 `break`、`goto` 或 `continue` 从 lambda 表达式跳到 lambda 表达式块外面的目标。
- 不能在 lambda 表达式内执行不安全的代码。
- 不能在 `is` 运算符的左边使用 lambda 表达式。
- 因为 lambda 表达式是匿名方法的超集，所以适用于匿名方法的所有限制也适用于 lambda 表达式。

4.12.4　参考

MSDN 文档中的“lambda 表达式（C# 编程指南）”主题。

4.13 在lambda表达式中使用不同的参数修饰符

4.13.1 问题

你知道可以把参数传递给 lambda 表达式，但是还需要确定可以对它们使用哪些有效的参数修饰符。

4.13.2 解决方案

lambda 表达式可以在它们的参数列表中使用 out 和 ref 参数修饰符，但是不能使用 params 修饰符。不过，这不会妨碍利用其中的任何修饰符来创建委托。

```
// 声明out委托
delegate int DoOutWork(out string work);

// 声明ref委托
delegate int DoRefWork(ref string work);

// 声明params委托
delegate int DoParamsWork(params object[] workItems);
```

即使在定义 DoParamsWork 委托时对参数使用 params 关键字，仍然可以把它用作 lambda 表达式的类型，稍后就会看到这一点。为了使用 DoOutWork 委托，可以使用 out 关键字创建一个内联 lambda 表达式，并把它赋予 DoOutWork 委托实例。在 lambda 表达式体内，首先给 out 变量赋值（因为通过定义为 out 参数，它并没有初始值），把它写到控制台，并返回字符串散列代码。注意在参数列表中，必须提供 s 的类型（即 string），因为无法推断标记有 out 或 ref 关键字的变量的类型。编译器不会推断 out 或 ref 变量，以保留调用站点和参数声明站点的类型表示，从而帮助开发人员清楚地推断这些变量的可能赋值。

```
// 声明实例并赋予方法
DoOutWork dow = (out string s) =>
{
    s = "WorkFinished";
    Console.WriteLine(s);
    return s.GetHashCode();
};
```

要运行 lambda 表达式代码，可以用一个 out 参数调用委托，然后把结果输出到控制台。

```
// 调用委托
string work;
int i = dow(out work);
Console.WriteLine(work);
```

为了在 lambda 表达式中使用 ref 参数修饰符，可以创建一个内联方法，利用 ref 参数挂接到 DoRefWork 委托。在该方法中，输出变量的原始值，重新赋值，并且获得新值的散列代码。记住，与 out 关键字一样，在参数列表中必须提供 s 的类型（即 string），因为无法推断标记有 ref 关键字的变量的类型。

```
// 声明实例并赋予方法
DoRefWork drw = (ref string s) =>
{
    Console.WriteLine(s);
    s = "WorkFinished";
    return s.GetHashCode();
};
```

要运行 lambda 表达式，给字符串 work 赋值，然后将其作为 ref 参数传递给实例化的 DoRefWork 委托。当委托调用返回时，输出 work 字符串的新值。

```
// 调用委托
work = "WorkStarted";
i = drw(ref work);
Console.WriteLine(work);
```

虽然可以利用 params 修饰符声明委托，但是不能在参数列表中利用 params 关键字使用 lambda 表达式挂接委托。如果试图这样做，编译器将在 DoParamsWork 行上显示编译错误："CS1670 params is not valid in this context"。

```
////作为lambda来使用,将会得到
////CS1670 "params is not valid in this context"
//DoParamsWork dpwl = (params object[] workItems) =>
//{
//    foreach (object o in workItems)
//    {
//        Console.WriteLine(o.ToString());
//    }
//    return workItems.GetHashCode();
//};
```

即使尝试使用匿名方法代替 lambda 表达式执行该操作，仍然不能在参数列表中利用 params 关键字挂接委托。如果试图这样做，编译器仍然将在 DoParamsWork 行上显示编译错误："CS1670 params is not valid in this context"。

```
//作为匿名方法来使用,将会得到
//CS1670 "params is not valid in this context"
//DoParamsWork dpwa = delegate (params object[] workItems)
//{
//    foreach (object o in workItems)
//    {
//        Console.WriteLine(o.ToString());
//    }
//    return workItems.GetHashCode();
//};
```

不过，可以省略 params 关键字，仍然为委托建立 lambda 表达式，如下所示。

```
// 我们能做的是省略params关键字
DoParamsWork dpw = workItems =>
{
    foreach (object o in workItems)
        Console.WriteLine(o.ToString());
    return workItems.GetHashCode();
};
```

我们注意到，尽管从 lambda 表达式中删除了 `params` 关键字，但这不会阻止你使用相同的语法。`params` 关键字存在于委托类型上，因此可以像下面这样调用它。

```
int i = dpw("Hello", "42", "bar");
```

这说明可以把 lambda 表达式绑定到使用 `params` 声明的委托上。一旦执行了该操作，就可以调用 lambda 表达式，按你所期望的那样传入任意数量的参数。

4.13.3 讨论

lambda 表达式不能访问外部作用域的 `ref` 或 `out` 参数。这意味着定义为包含方法的一部分的任何 `out` 或 `ref` 变量都禁止在 lambda 表达式体内部使用。

```
public void TestOut(out string outStr)
{
    // 声明实例
    DoWork dw = s =>
    {
        Console.WriteLine(s);
        // 导致错误CS1628：
        // "Cannot use ref or out parameter 'outStr' inside an
        // anonymous method, lambda expression, or query expression"
        outStr = s;
        return s.GetHashCode();
    };
    // 调用委托
    int i = dw("DoWorkMethodImpl1");
}

public void TestRef(ref string refStr)
{
    // 声明实例
    DoWork dw = s =>
    {
        Console.WriteLine(s);
        // 导致错误 CS1628：
        // "Cannot use ref or out parameter 'refStr' inside an
        // anonymous method, lambda expression, or query expression"
        refStr = s;
        return s.GetHashCode();
    };
    // 调用委托
    int i = dw("DoWorkMethodImpl1");
}
```

非常有趣的是，lambda 表达式可以访问带有 `params` 修饰符的外层变量。

```
// 声明委托
delegate int DoWork(string work);

public void TestParams(params string[] items)
{
    // 声明实例
    DoWork dw = s =>
```

```
        {
            Console.WriteLine(s);
            foreach (string item in items)
                Console.WriteLine(item);
            return s.GetHashCode();
        };
        // 调用委托
        int i = dw("DoWorkMethodImpl1");
    }
```

由于 `params` 修饰符可以给调用站点提供好处（因此编译器知道使之成为一个支持可变长度的参数列表的方法调用），并且它永远不会直接调用 lambda 表达式（总是通过委托调用），那么出于给调用站点提供好处的目的对 lambda 表达式进行某种修饰就显得毫无意义——毕竟没有调用站点。因此，不能对 lambda 表达式使用 `params` 关键字是无关紧要的。对于 lambda 表达式，调用站点总是通过委托调用它，因此委托是否具有 `params` 关键字才是要紧的事。

4.13.4　参考

范例 1.17（即 1.17 节）；MSDN 文档中的“CS1670”“CS1525”“CS1628”“`out`”“`ref`”“`params`”和“`System.ParamArrayAttribute`”主题。

4.14　用并行来加速LINQ操作

4.14.1　问题

一个执行代价高昂操作的 LINQ 查询拖慢了整个处理，你希望能提升它的速度。

4.14.2　解决方案

使用并行 LINQ（parallel LINQ，PLINQ）来利用机器的全部性能更快地处理查询。

要演示这一点，让我们来考虑一下 Brooke 和 Katie 的困境。Brooke 和 Katie 正一起忙于一本烹饪手册，他们需要评估所有章节中的所有食谱。因为有非常多的食谱，所以他们想要将食谱基本的验证步骤交出去，然后由 Brooke 或者 Katie 作为主编对每个食谱进行最后的完善工作。

每个 `Chapter` 包含一些 `Recipe`，`Recipe` 的验证步骤包括以下四个。

(1) 阅读食谱的文本作为前提。
(2) 检查食谱的原料和份量。
(3) 准备食谱，将食谱的每个难度级别品尝一次。
(4) 由 Brooke 或 Kaite 完成最终的编辑步骤。

如果食谱评估的任何阶段未能通过，这一阶段就需要重做，品尝阶段除外。如果食谱未能通过品尝阶段，需要从头开始。

要使用常规的 LINQ 来处理 RecipeChapter 的集合（例子中的 chapters），可以使用以下语句。

```
chapters.Select(c => TimedEvaluateChapter(c, rnd)).ToList();
```

TimedEvaluateChapter 方法对 RecipeChapter 和其中的所有 Recipe 进行评估，同时对评估过程计时。对 RecipeChapter 中的每一个 Recipe 都调用一次 EvaluateRecipe 以执行 Recipe 的检验步骤。

```
private static RecipeChapter TimedEvaluateChapter(RecipeChapter rc, Random rnd)
{
    Stopwatch watch = new Stopwatch();
    LogOutput($"Evaluating Chapter {rc}");
    watch.Start();
    foreach (var r in rc.Recipes)
        EvaluateRecipe(r, rnd);
    watch.Stop();
    LogOutput($"Finished Evaluating Chapter {rc}");
    return rc;
}
```

为了更快速地处理 Recipe，我们在调用 Select 为每一个 RecipeChapter 调用 TimedEvaluateChapter 之前，添加了对 AsParallel 扩展方法的调用。

```
chapters.AsParallel().Select(c => TimedEvaluateChapter(c, rnd)).ToList();
```

运行结果取决于你的硬件，但下面的结果记录了一次先运行常规 LINQ，然后运行 PLINQ 的计时。

```
Full Chapter Evaluation with LINQ took: 00:01:19.1395258
Full Chapter Evaluation with PLINQ took: 00:00:25.1708103
```

4.14.3 讨论

当使用 PLINQ 时，要记得的首要大事是并行处理的工作单元数量要足够大，以抵消并行的成本。并行操作有一些额外的设置和拆解成本（例如将数据集分区），如果数据集太小或者在每个成员上的操作代价并不高，将不足以从并行技术中受益，实际性能可能会更差一些。如果 PLINQ 确定它并不能高效地将查询并行，它将会顺序处理查询。当此情况发生时，根据你的特定情形（WithExecutionMode、WithDegreeOfParallelism），可以使用另外一些额外的方法进行调整。

与所有的工程问题一样，测量你的结果是理解是否有所提升的关键。考虑到这一点，我们创建了 TimedEvaluateChapter 方法以在 Select 语句中调用。

```
chapters.AsParallel().Select(c => TimedEvaluateChapter(c, rnd)).ToList();
```

TimedEvaluateChapter 对评估 RecipeChapter 中所有 Recipe 的过程进行了计时，调用了 Stopwatch.Start 和 Stopwatch.Stop 实现计时值的包装。注意，如果你没有调用 Stopwatch.Reset 就重新启动了 Stopwatch，那么计时结果将累加到 Stopwatch 中已有的值上，你也许就得到比预期更大一些的值。

```
private static RecipeChapter TimedEvaluateChapter(RecipeChapter rc, Random rnd)
{
    Stopwatch watch = new Stopwatch();
    LogOutput($"Evaluating Chapter {rc}");
    watch.Start();
    foreach (var r in rc.Recipes)
        EvaluateRecipe(r, rnd);
    watch.Stop();
    LogOutput($"Finished Evaluating Chapter {rc}");
    return rc;
}
```

EvaluateRecipe 递归执行每个食谱的验证步骤，直到通过了 Brooke 和 Katie 的最终编辑。调用 Thread.Sleep 以模拟每个步骤的工作。

```
private static Recipe EvaluateRecipe(Recipe r, Random rnd)
{
    //食谱编辑步骤
    if (!r.TextApproved)
    {
        //阅读食谱以确定是否合理
        Thread.Sleep(50);
        int evaluation = rnd.Next(1, 10);
        // 7表示不合理,不批准,
        // 打回重做
        if (evaluation == 7)
        {
            LogOutput($"{r} failed the readthrough! Reworking...");
        }
        else
            r.TextApproved = true;
        return EvaluateRecipe(r, rnd);
    }
    else if (!r.IngredientsApproved)
    {
        //检查原料和份量
        Thread.Sleep(100);
        int evaluation = rnd.Next(1, 10);
        // 3表示原料或份量不对,
        // 打回重做
        if (evaluation == 3)
        {
            LogOutput($"{r} had incorrect measurements! Reworking...");
        }
        else
            r.IngredientsApproved = true;
        return EvaluateRecipe(r, rnd);
    }
    else if (r.RecipeEvaluated != r.Rank)
    {
        //准备食谱和品尝
        Thread.Sleep(50 * r.Rank);
        int evaluation = rnd.Next(1, 10);
        // 4表示尝起来不对,打回重做
        if (evaluation == 4)
```

```
            {
                r.TextApproved = false;
                r.IngredientsApproved = false;
                r.RecipeEvaluated = 0;
                LogOutput($"{r} tasted bad!  Reworking...");
            }
            else
                r.RecipeEvaluated++;
            return EvaluateRecipe(r, rnd);
        }
        else
        {
            //最终的编辑阶段(Brooke或Katie)
            Thread.Sleep(50 * r.Rank);
            int evaluation = rnd.Next(1, 10);
            // 1表示食谱并不完善,打回重做
            if (evaluation == 1)
            {
                r.TextApproved = false;
                r.IngredientsApproved = false;
                r.RecipeEvaluated = 0;
                LogOutput($"{r} failed final editing!  Reworking...");
                return EvaluateRecipe(r, rnd);
            }
            else
            {
                r.FinalEditingComplete = true;
                LogOutput($"{r} is ready for release!");
            }
        }
        return r;
    }
```

下列是 RecipeChapter 和 Recipe 类的定义，用于帮助 Brooke 和 Katie 评估所有食谱。

```
    public class RecipeChapter
    {
        public int Number { get; set; }
        public string Title { get; set; }
        public List<Recipe> Recipes { get; set; }
        public override string ToString() => $"{Number} - {Title}";
    }

    public class Recipe
    {
        public RecipeChapter Chapter { get; set; }
        public string MainIngredient { get; set; }
        public int Number { get; set; }
        public bool TextApproved { get; set; }
        public bool IngredientsApproved { get; set; }

        /// <summary>
        // Recipe应该评估的次数与食谱的Rank值相同
        /// </summary>
        public int RecipeEvaluated { get; set; }
```

```
    public bool FinalEditingComplete { get; set; }

    public int Rank { get; set; }

    public override string ToString() =>
        $"{Chapter.Number}.{Number} ({Chapter.Title}:{MainIngredient})";
}
```

LINQ 的输出样例如下所示，它顺序地处理集合。

```
Running Cookbook Evaluation
Evaluating Chapter 1 - Soups
1.1 (Soups:Sprouts, Mung Bean) is ready for release!
1.2 (Soups:Potato Bread) is ready for release!
1.3 (Soups:Chicken Liver) tasted bad!  Reworking...
1.3 (Soups:Chicken Liver) is ready for release!
1.4 (Soups:Cherimoya) tasted bad!  Reworking...
1.4 (Soups:Cherimoya) had incorrect measurements! Reworking...
1.4 (Soups:Cherimoya) is ready for release!
1.5 (Soups:High-Protein Bread) is ready for release!
1.6 (Soups:Flat Bread) failed the readthrough! Reworking...
1.6 (Soups:Flat Bread) is ready for release!
1.7 (Soups:Pomegranate) is ready for release!
1.8 (Soups:Carissa, Natal Plum) had incorrect measurements! Reworking...
1.8 (Soups:Carissa, Natal Plum) is ready for release!
1.9 (Soups:Ideal Flat Bread) is ready for release!
1.10 (Soups:Banana Bread) tasted bad!  Reworking...
1.10 (Soups:Banana Bread) is ready for release!
Finished Evaluating Chapter 1 - Soups
Evaluating Chapter 2 - Salads
2.1 (Salads:Caraway) tasted bad!  Reworking...
2.1 (Salads:Caraway) tasted bad!  Reworking...
2.1 (Salads:Caraway) had incorrect measurements! Reworking...
2.1 (Salads:Caraway) is ready for release!
2.2 (Salads:Potatoes, Red) had incorrect measurements! Reworking...
2.2 (Salads:Potatoes, Red) tasted bad!  Reworking...
2.2 (Salads:Potatoes, Red) is ready for release!
2.3 (Salads:Lemon) is ready for release!
2.4 (Salads:Cream cheese) is ready for release!
2.5 (Salads:Artichokes, Domestic) is ready for release!
2.6 (Salads:Grapefruit) is ready for release!
2.7 (Salads:Lettuce, Iceberg) is ready for release!
2.8 (Salads:Fenugreek) is ready for release!
2.9 (Salads:Ostrich) is ready for release!
2.10 (Salads:Brazil Nuts) tasted bad!  Reworking...
2.10 (Salads:Brazil Nuts) had incorrect measurements! Reworking...
2.10 (Salads:Brazil Nuts) tasted bad!  Reworking...
2.10 (Salads:Brazil Nuts) is ready for release!
Finished Evaluating Chapter 2 - Salads
Evaluating Chapter 3 - Appetizers
3.1 (Appetizers:Loquat) tasted bad!  Reworking...
3.1 (Appetizers:Loquat) had incorrect measurements! Reworking...
3.1 (Appetizers:Loquat) tasted bad!  Reworking...
3.1 (Appetizers:Loquat) is ready for release!
```

```
3.2 (Appetizers:Bergenost) is ready for release!
3.3 (Appetizers:Tomato Red Roma) had incorrect measurements! Reworking...
3.3 (Appetizers:Tomato Red Roma) tasted bad!  Reworking...
3.3 (Appetizers:Tomato Red Roma) tasted bad!  Reworking...
3.3 (Appetizers:Tomato Red Roma) is ready for release!
3.4 (Appetizers:Guava) failed final editing!  Reworking...
3.4 (Appetizers:Guava) is ready for release!
3.5 (Appetizers:Squash Flower) is ready for release!
3.6 (Appetizers:Radishes, Red) is ready for release!
3.7 (Appetizers:Goose Liver) tasted bad!  Reworking...
3.7 (Appetizers:Goose Liver) had incorrect measurements! Reworking...
3.7 (Appetizers:Goose Liver) is ready for release!
3.8 (Appetizers:Okra) had incorrect measurements! Reworking...
3.8 (Appetizers:Okra) is ready for release!
3.9 (Appetizers:Borage) is ready for release!
3.10 (Appetizers:Peppers) is ready for release!
Finished Evaluating Chapter 3 - Appetizers
Evaluating Chapter 4 - Entrees
4.1 (Entrees:Plantain) is ready for release!
4.2 (Entrees:Pignola (Pine)) is ready for release!
4.3 (Entrees:Potatoes, Gold) is ready for release!
4.4 (Entrees:Ribeye) failed the readthrough! Reworking...
4.4 (Entrees:Ribeye) is ready for release!
4.5 (Entrees:Sprouts, Mung Bean) failed the readthrough! Reworking...
4.5 (Entrees:Sprouts, Mung Bean) had incorrect measurements! Reworking...
4.5 (Entrees:Sprouts, Mung Bean) failed final editing!  Reworking...
4.5 (Entrees:Sprouts, Mung Bean) is ready for release!
4.6 (Entrees:Squash) had incorrect measurements! Reworking...
4.6 (Entrees:Squash) is ready for release!
4.7 (Entrees:Squash, Winter) tasted bad!  Reworking...
4.7 (Entrees:Squash, Winter) is ready for release!
4.8 (Entrees:Corn, Blue) is ready for release!
4.9 (Entrees:Snake) had incorrect measurements! Reworking...
4.9 (Entrees:Snake) tasted bad!  Reworking...
4.9 (Entrees:Snake) tasted bad!  Reworking...
4.9 (Entrees:Snake) is ready for release!
4.10 (Entrees:Prosciutto) is ready for release!
Finished Evaluating Chapter 4 - Entrees
Evaluating Chapter 5 - Desserts
5.1 (Desserts:Mushroom, White, Silver Dollar) tasted bad!  Reworking...
5.1 (Desserts:Mushroom, White, Silver Dollar) had incorrect measurements!
Reworking...
5.1 (Desserts:Mushroom, White, Silver Dollar) tasted bad!  Reworking...
5.1 (Desserts:Mushroom, White, Silver Dollar) tasted bad!  Reworking...
5.1 (Desserts:Mushroom, White, Silver Dollar) had incorrect measurements!
Reworking...
5.1 (Desserts:Mushroom, White, Silver Dollar) is ready for release!
5.2 (Desserts:Eggplant) is ready for release!
5.3 (Desserts:Asparagus Peas) tasted bad!  Reworking...
5.3 (Desserts:Asparagus Peas) failed the readthrough! Reworking...
5.3 (Desserts:Asparagus Peas) failed the readthrough! Reworking...
5.3 (Desserts:Asparagus Peas) is ready for release!
5.4 (Desserts:Squash, Kabocha) failed the readthrough! Reworking...
5.4 (Desserts:Squash, Kabocha) tasted bad!  Reworking...
5.4 (Desserts:Squash, Kabocha) is ready for release!
```

```
5.5 (Desserts:Sprouts, Radish) is ready for release!
5.6 (Desserts:Mushroom, Black Trumpet) is ready for release!
5.7 (Desserts:Tea Cakes) tasted bad!  Reworking...
5.7 (Desserts:Tea Cakes) tasted bad!  Reworking...
5.7 (Desserts:Tea Cakes) failed the readthrough! Reworking...
5.7 (Desserts:Tea Cakes) is ready for release!
5.8 (Desserts:Blueberries) had incorrect measurements! Reworking...
5.8 (Desserts:Blueberries) tasted bad!  Reworking...
5.8 (Desserts:Blueberries) is ready for release!
5.9 (Desserts:Sago Palm) is ready for release!
5.10 (Desserts:Opossum) had incorrect measurements! Reworking...
5.10 (Desserts:Opossum) is ready for release!
Finished Evaluating Chapter 5 - Desserts
Evaluating Chapter 6 - Snacks
6.1 (Snacks:Cheddar) tasted bad!  Reworking...
6.1 (Snacks:Cheddar) is ready for release!
6.2 (Snacks:Melon, Bitter) is ready for release!
6.3 (Snacks:Scallion) is ready for release!
6.4 (Snacks:Squash Chayote) failed final editing!  Reworking...
6.4 (Snacks:Squash Chayote) is ready for release!
6.5 (Snacks:Roasted Turkey) is ready for release!
6.6 (Snacks:Lime) is ready for release!
6.7 (Snacks:Hazelnut) is ready for release!
6.8 (Snacks:Radishes, Daikon) tasted bad!  Reworking...
6.8 (Snacks:Radishes, Daikon) tasted bad!  Reworking...
6.8 (Snacks:Radishes, Daikon) failed the readthrough! Reworking...
6.8 (Snacks:Radishes, Daikon) tasted bad!  Reworking...
6.8 (Snacks:Radishes, Daikon) is ready for release!
6.9 (Snacks:Salami) failed the readthrough! Reworking...
6.9 (Snacks:Salami) is ready for release!
6.10 (Snacks:Mushroom, Oyster) failed the readthrough! Reworking...
6.10 (Snacks:Mushroom, Oyster) is ready for release!
Finished Evaluating Chapter 6 - Snacks
Evaluating Chapter 7 - Breakfast
7.1 (Breakfast:Daikon Radish) had incorrect measurements! Reworking...
7.1 (Breakfast:Daikon Radish) is ready for release!
7.2 (Breakfast:Lettuce, Red Leaf) failed final editing!  Reworking...
7.2 (Breakfast:Lettuce, Red Leaf) is ready for release!
7.3 (Breakfast:Alfalfa Sprouts) is ready for release!
7.4 (Breakfast:Tea Cakes) is ready for release!
7.5 (Breakfast:Chia seed) is ready for release!
7.6 (Breakfast:Tangerine) is ready for release!
7.7 (Breakfast:Spinach) is ready for release!
7.8 (Breakfast:Flank Steak) is ready for release!
7.9 (Breakfast:Loganberries) had incorrect measurements! Reworking...
7.9 (Breakfast:Loganberries) had incorrect measurements! Reworking...
7.9 (Breakfast:Loganberries) had incorrect measurements! Reworking...
7.9 (Breakfast:Loganberries) is ready for release!
7.10 (Breakfast:Opossum) is ready for release!
Finished Evaluating Chapter 7 - Breakfast
Evaluating Chapter 8 - Sandwiches
8.1 (Sandwiches:Rhubarb) tasted bad!  Reworking...
8.1 (Sandwiches:Rhubarb) is ready for release!
8.2 (Sandwiches:Pickle, Brine) is ready for release!
8.3 (Sandwiches:Oranges) tasted bad!  Reworking...
```

```
8.3 (Sandwiches:Oranges) had incorrect measurements! Reworking...
8.3 (Sandwiches:Oranges) is ready for release!
8.4 (Sandwiches:Chayote, Pipinella, Vegetable Pear) tasted bad!  Reworking...
8.4 (Sandwiches:Chayote, Pipinella, Vegetable Pear) is ready for release!
8.5 (Sandwiches:Bear) is ready for release!
8.6 (Sandwiches:Panela) had incorrect measurements! Reworking...
8.6 (Sandwiches:Panela) is ready for release!
8.7 (Sandwiches:Peppers, Red) had incorrect measurements! Reworking...
8.7 (Sandwiches:Peppers, Red) tasted bad!  Reworking...
8.7 (Sandwiches:Peppers, Red) failed the readthrough! Reworking...
8.7 (Sandwiches:Peppers, Red) failed the readthrough! Reworking...
8.7 (Sandwiches:Peppers, Red) had incorrect measurements! Reworking...
8.7 (Sandwiches:Peppers, Red) tasted bad!  Reworking...
8.7 (Sandwiches:Peppers, Red) is ready for release!
8.8 (Sandwiches:Oat Bread) is ready for release!
8.9 (Sandwiches:Peppers, Green) is ready for release!
8.10 (Sandwiches:Garlic) is ready for release!
Finished Evaluating Chapter 8 - Sandwiches
***********************************************
Full Chapter Evaluation with LINQ took: 00:01:19.1395258
***********************************************
```

PLINQ 的输出示例如下所示，它是并行处理的（注意开头处对 4 个 `RecipeChapter` 进行了评估），以打乱的顺序处理数据项。

```
Evaluating Chapter 5 - Desserts
Evaluating Chapter 3 - Appetizers
Evaluating Chapter 1 - Soups
Evaluating Chapter 7 - Breakfast
7.1 (Breakfast:Daikon Radish) failed the readthrough! Reworking...
1.1 (Soups:Sprouts, Mung Bean) failed the readthrough! Reworking...
3.1 (Appetizers:Loquat) had incorrect measurements! Reworking...
1.1 (Soups:Sprouts, Mung Bean) had incorrect measurements! Reworking...
7.1 (Breakfast:Daikon Radish) tasted bad!  Reworking...
5.1 (Desserts:Mushroom, White, Silver Dollar) tasted bad!  Reworking...
3.1 (Appetizers:Loquat) failed final editing!  Reworking...
7.1 (Breakfast:Daikon Radish) is ready for release!
3.1 (Appetizers:Loquat) tasted bad!  Reworking...
5.1 (Desserts:Mushroom, White, Silver Dollar) tasted bad!  Reworking...
1.1 (Soups:Sprouts, Mung Bean) is ready for release!
3.1 (Appetizers:Loquat) is ready for release!
1.2 (Soups:Potato Bread) had incorrect measurements! Reworking...
1.2 (Soups:Potato Bread) is ready for release!
1.3 (Soups:Chicken Liver) failed the readthrough! Reworking...
3.2 (Appetizers:Bergenost) is ready for release!
1.3 (Soups:Chicken Liver) had incorrect measurements! Reworking...
7.2 (Breakfast:Lettuce, Red Leaf) failed final editing!  Reworking...
5.1 (Desserts:Mushroom, White, Silver Dollar) is ready for release!
5.2 (Desserts:Eggplant) is ready for release!
7.2 (Breakfast:Lettuce, Red Leaf) tasted bad!  Reworking...
3.3 (Appetizers:Tomato Red Roma) is ready for release!
1.3 (Soups:Chicken Liver) is ready for release!
3.4 (Appetizers:Guava) is ready for release!
5.3 (Desserts:Asparagus Peas) is ready for release!
1.4 (Soups:Cherimoya) is ready for release!
```

```
5.4 (Desserts:Squash, Kabocha) is ready for release!
1.5 (Soups:High-Protein Bread) had incorrect measurements! Reworking...
7.2 (Breakfast:Lettuce, Red Leaf) failed final editing!  Reworking...
1.5 (Soups:High-Protein Bread) failed final editing!  Reworking...
5.5 (Desserts:Sprouts, Radish) is ready for release!
3.5 (Appetizers:Squash Flower) is ready for release!
3.6 (Appetizers:Radishes, Red) failed the readthrough! Reworking...
1.5 (Soups:High-Protein Bread) is ready for release!
5.6 (Desserts:Mushroom, Black Trumpet) tasted bad!  Reworking...
1.6 (Soups:Flat Bread) is ready for release!
1.7 (Soups:Pomegranate) is ready for release!
3.6 (Appetizers:Radishes, Red) is ready for release!
7.2 (Breakfast:Lettuce, Red Leaf) is ready for release!
5.6 (Desserts:Mushroom, Black Trumpet) failed final editing!  Reworking...
1.8 (Soups:Carissa, Natal Plum) is ready for release!
7.3 (Breakfast:Alfalfa Sprouts) is ready for release!
7.4 (Breakfast:Tea Cakes) is ready for release!
5.6 (Desserts:Mushroom, Black Trumpet) is ready for release!
3.7 (Appetizers:Goose Liver) is ready for release!
1.9 (Soups:Ideal Flat Bread) is ready for release!
5.7 (Desserts:Tea Cakes) tasted bad!  Reworking...
3.8 (Appetizers:Okra) is ready for release!
3.9 (Appetizers:Borage) tasted bad!  Reworking...
3.9 (Appetizers:Borage) failed the readthrough! Reworking...
3.9 (Appetizers:Borage) failed the readthrough! Reworking...
7.5 (Breakfast:Chia seed) is ready for release!
3.9 (Appetizers:Borage) is ready for release!
1.10 (Soups:Banana Bread) is ready for release!
Finished Evaluating Chapter 1 - Soups
Evaluating Chapter 2 - Salads
3.10 (Appetizers:Peppers) is ready for release!
Finished Evaluating Chapter 3 - Appetizers
Evaluating Chapter 4 - Entrees
5.7 (Desserts:Tea Cakes) is ready for release!
7.6 (Breakfast:Tangerine) is ready for release!
4.1 (Entrees:Plantain) is ready for release!
4.2 (Entrees:Pignola (Pine)) failed the readthrough! Reworking...
2.1 (Salads:Caraway) is ready for release!
5.8 (Desserts:Blueberries) is ready for release!
5.9 (Desserts:Sago Palm) failed the readthrough! Reworking...
5.9 (Desserts:Sago Palm) tasted bad!  Reworking...
5.9 (Desserts:Sago Palm) is ready for release!
4.2 (Entrees:Pignola (Pine)) is ready for release!
2.2 (Salads:Potatoes, Red) is ready for release!
2.3 (Salads:Lemon) had incorrect measurements! Reworking...
4.3 (Entrees:Potatoes, Gold) is ready for release!
7.7 (Breakfast:Spinach) failed final editing!  Reworking...
2.3 (Salads:Lemon) had incorrect measurements! Reworking...
4.4 (Entrees:Ribeye) had incorrect measurements! Reworking...
7.7 (Breakfast:Spinach) tasted bad!  Reworking...
4.4 (Entrees:Ribeye) is ready for release!
2.3 (Salads:Lemon) tasted bad!  Reworking...
5.10 (Desserts:Opossum) is ready for release!
Finished Evaluating Chapter 5 - Desserts
Evaluating Chapter 6 - Snacks
```

```
6.1 (Snacks:Cheddar) is ready for release!
4.5 (Entrees:Sprouts, Mung Bean) is ready for release!
7.7 (Breakfast:Spinach) is ready for release!
6.2 (Snacks:Melon, Bitter) is ready for release!
6.3 (Snacks:Scallion) failed the readthrough! Reworking...
7.8 (Breakfast:Flank Steak) tasted bad!  Reworking...
2.3 (Salads:Lemon) failed final editing!  Reworking...
7.8 (Breakfast:Flank Steak) is ready for release!
4.6 (Entrees:Squash) is ready for release!
2.3 (Salads:Lemon) tasted bad!  Reworking...
4.7 (Entrees:Squash, Winter) failed the readthrough! Reworking...
4.7 (Entrees:Squash, Winter) had incorrect measurements! Reworking...
6.3 (Snacks:Scallion) is ready for release!
6.4 (Snacks:Squash Chayote) is ready for release!
4.7 (Entrees:Squash, Winter) is ready for release!
7.9 (Breakfast:Loganberries) is ready for release!
2.3 (Salads:Lemon) is ready for release!
7.10 (Breakfast:Opossum) is ready for release!
Finished Evaluating Chapter 7 - Breakfast
Evaluating Chapter 8 - Sandwiches
8.1 (Sandwiches:Rhubarb) had incorrect measurements! Reworking...
4.8 (Entrees:Corn, Blue) is ready for release!
2.4 (Salads:Cream cheese) failed final editing!  Reworking...
2.4 (Salads:Cream cheese) is ready for release!
6.5 (Snacks:Roasted Turkey) failed final editing!  Reworking...
4.9 (Entrees:Snake) is ready for release!
4.10 (Entrees:Prosciutto) failed the readthrough! Reworking...
6.5 (Snacks:Roasted Turkey) had incorrect measurements! Reworking...
2.5 (Salads:Artichokes, Domestic) tasted bad!  Reworking...
4.10 (Entrees:Prosciutto) tasted bad!  Reworking...
8.1 (Sandwiches:Rhubarb) tasted bad!  Reworking...
4.10 (Entrees:Prosciutto) had incorrect measurements! Reworking...
4.10 (Entrees:Prosciutto) is ready for release!
Finished Evaluating Chapter 4 - Entrees
6.5 (Snacks:Roasted Turkey) is ready for release!
6.6 (Snacks:Lime) had incorrect measurements! Reworking...
2.5 (Salads:Artichokes, Domestic) failed final editing!  Reworking...
8.1 (Sandwiches:Rhubarb) is ready for release!
6.6 (Snacks:Lime) tasted bad!  Reworking...
6.6 (Snacks:Lime) is ready for release!
2.5 (Salads:Artichokes, Domestic) is ready for release!
6.7 (Snacks:Hazelnut) is ready for release!
8.2 (Sandwiches:Pickle, Brine) is ready for release!
2.6 (Salads:Grapefruit) is ready for release!
2.7 (Salads:Lettuce, Iceberg) failed final editing!  Reworking...
2.7 (Salads:Lettuce, Iceberg) is ready for release!
6.8 (Snacks:Radishes, Daikon) is ready for release!
8.3 (Sandwiches:Oranges) is ready for release!
6.9 (Snacks:Salami) tasted bad!  Reworking...
2.8 (Salads:Fenugreek) is ready for release!
8.4 (Sandwiches:Chayote, Pipinella, Vegetable Pear) tasted bad!  Reworking...
2.9 (Salads:Ostrich) failed the readthrough! Reworking...
6.9 (Snacks:Salami) is ready for release!
6.10 (Snacks:Mushroom, Oyster) is ready for release!
Finished Evaluating Chapter 6 - Snacks
```

```
2.9 (Salads:Ostrich) failed final editing!  Reworking...
2.9 (Salads:Ostrich) failed the readthrough! Reworking...
2.9 (Salads:Ostrich) failed the readthrough! Reworking...
8.4 (Sandwiches:Chayote, Pipinella, Vegetable Pear) is ready for release!
8.5 (Sandwiches:Bear) is ready for release!
2.9 (Salads:Ostrich) failed final editing!  Reworking...
8.6 (Sandwiches:Panela) tasted bad!  Reworking...
8.6 (Sandwiches:Panela) failed the readthrough! Reworking...
2.9 (Salads:Ostrich) is ready for release!
8.6 (Sandwiches:Panela) had incorrect measurements! Reworking...
8.6 (Sandwiches:Panela) is ready for release!
2.10 (Salads:Brazil Nuts) is ready for release!
Finished Evaluating Chapter 2 - Salads
8.7 (Sandwiches:Peppers, Red) tasted bad!  Reworking...
8.7 (Sandwiches:Peppers, Red) tasted bad!  Reworking...
8.7 (Sandwiches:Peppers, Red) is ready for release!
8.8 (Sandwiches:Oat Bread) is ready for release!
8.9 (Sandwiches:Peppers, Green) is ready for release!
8.10 (Sandwiches:Garlic) is ready for release!
Finished Evaluating Chapter 8 - Sandwiches
***********************************************
Full Chapter Evaluation with PLINQ took: 00:00:25.1708103
***********************************************
Cookbook Evaluation Complete
```

如果你运行了一个 PLINQ 查询，并且调用到的操作引发了一个异常，此时它将不会停止运算，而是继续运行，并将所有异常记录到一个 `AggregateException` 中。这个 `AggregateException` 可以在查询运算求值后捕获到（而非查询声明后）。

4.14.4　参考

MSDN 文档中的“并行 LINQ”主题。

第 5 章

调试和异常处理

5.0 简介

本章包含的范例涉及异常处理机制，包括 `try`、`catch` 和 `finally` 语句块。除了这些范例之外，还有一些范例介绍了用于从代码内手动引发异常的机制。最后的范例涉及 `Exception` 类及其使用，以及对其进行子类化以创建新的异常类型。

通常，异常处理的设计和实现是在开发周期后期执行的。但是，由于 C# 异常处理的能力和复杂性，需要更早地计划乃至实现异常处理模式。这样做可以提高代码的可靠性和健壮性，同时可以将在绝大部分或全部应用程序编码完成之后添加异常处理的影响减至最小。

C# 中的异常处理非常灵活。它允许选择一种细粒度或粗粒度的方法进行错误处理，任意中间的处理粒度也均可。这意味着可以在任何一行代码周围（细粒度方法）或者在一个调用其他许多方法的方法周围（粗粒度方法）添加异常处理，也可以混合使用这两种方法：主要使用粗粒度方法，并且在特定的关键代码区域使用更细粒度的方法。在使用细粒度方法时，可以截获可能只由几行代码引发的特定异常。下面的方法使用细粒度的异常处理把对象的属性设置为一个数值。

```
protected void SetValue(object value)
{
    try
    {
        myObj.Property1 = value;
    }
    catch (NullReferenceException)
    {
        // 在此处处理这一调用可能引发的异常
    }
}
```

因此，如果在整个应用程序中使用这种方法，可能会增加大量多余的代码。如果只有一行或几行代码，并且需要以一种特定的方式处理异常，就应该使用这种细粒度的异常处理方法。如果无需在这种级别进行特定的错误处理，就应该把异常冒泡到栈上。例如，使用前面的 SetValue 方法，你可能必须通知用户发生的异常，并提供重试的机会。如果无论何时 myObj 的方法之一引发异常都需要调用 myObj 上的一个方法，就应该确保在合适的时间调用该方法。

粗粒度的异常处理完全相反，它使用更少的 try-catch 或 try-catch-finally 块。一个例子是在应用程序或组件的每个 public 方法中的所有代码周围放置一个 try-catch 块。这样做允许在代码中的最高级别上处理异常。如果在代码中的任意位置引发异常，都将把它冒泡到调用栈上，直至找到可以处理它的 catch 块为止。如果在所有公共方法上放置 try-catch 块，则会将所有异常冒泡给这个方法并处理它们。这样可以少写很多异常处理代码，但是会降低处理那些可能发生在代码中特定区域的特定异常的能力。你必须确定向应用程序中添加异常处理代码的最佳方式。这意味着在应用程序中的细粒度异常处理与粗粒度异常处理之间达到一种适当的平衡。

C# 允许不带任何参数地编写 catch 块。这样的一个示例如下所示。

```
public void CallCOMMethod()
{
    try
    {
        // Call a method on a COM object.
        myCOMObj.Method1();
    }
    catch
    {
        // 在此处处理这一调用可能引发的异常
    }
}
```

不带参数的 catch 继承自 C++。在 C++ 中，异常对象不必从 Exception 类派生而来。在 C++ 中以这种方式编写 catch 子句允许捕获作为异常引发的任何类型的对象。不过，在 C# 中，只有从 Exception 基类派生而来的任何对象才可能作为异常引发。使用不带参数的 catch 块允许捕获所有异常，但是将不能查看异常及其信息。以这种方式编写的 catch 块如下所示。

```
catch
{
     // 不能编写下一行所示的代码
     //Console.WriteLine(e.ToString);
}
```

它与下面这个 catch 块相同：

```
catch (Exception e)
{
     // 能够编写下一行所示的代码
     Console.WriteLine(e.ToString);
}
```

只不过在第二种情况下提供了异常参数，因此可以访问 Exception 对象。

不要编写不带任何参数的 catch 块，这样做将阻止你访问引发的实际 Exception 对象。

在 catch 块中捕获异常时，应该事先确定何时需要重新引发异常，何时需要把异常包装在一个外部异常中并引发它，以及何时应该立即处理异常并且不重新引发它。

当原始异常对调用者没有意义时，把异常包装在一个外部异常中是一个良好的惯例。当把异常包装在一个外部异常中时，需要确定什么异常最适合包装捕获的异常。一条经验法则是，包装异常应该总是便于跟踪原始问题，而不要用无关的或模糊的包装异常来屏蔽原始异常。在极少情况下，会认为屏蔽异常是合理的，其中有一种情况是，如果异常将要跨越信任界限，出于安全原因就必须屏蔽它。

捕获异常时另一个很有用的实践是，在代码中提供处理特定异常的 catch 块。记住基类异常（在 catch 块中使用时）不仅会捕获该类型，还会捕获其所有子类。

下面的代码使用特定的 catch 语句块以合适的方式处理不同的异常。

```
public void CallCOMMethod()
{
    try
    {
        // 调用COM对象上的一个方法
        myCOMObj.Method1();
    }
    catch (System.Runtime.InteropServices.ExternalException)
    {
        // 在此处处理这一调用可能引发的COM异常
    }
    catch (InvalidOperationException)
    {
        // 处理当前状态下针对COM对象的任何可能无效的方法调用
    }
}
```

在这段代码中，ExternalException 及其所派生异常的处理方式不同于 InvalidOperationException 及其派生的异常。如果从 myCOMObj.Method1 引发任何其他类型的异常，将不会在这里处理它们，但是会冒泡直至找到有效的 catch 块。如果没有找到有效的 catch 块，就会认为该异常未被处理，并且终止应用程序。

有时，不管是否引发异常，都必须执行清理代码。当引发异常时，任何对象都必须置于一种稳定的已知状态中。在这些情况下，如果必须执行代码，就使用 finally 语句块。下面的代码被修改成了使用 finally 语句块。

```
public void CallCOMMethod()
{
    try
    {
        // 调用COM对象上的一个方法
        myCOMObj.Method1();
    }
    catch (System.Runtime.InteropServices.ExternalException)
```

```
    {
        // 在此处处理这一调用可能引发的COM异常
    }
    finally
    {
        // 在此处清理并释放任何资源
        // 例如,在myCOMObj上可能有一个方法允许我们
        // 在使用Method1方法后进行清理
    }
}
```

> 不管 try 和 catch 语句块中发生什么事情，总会执行 finally 语句块。即使在 try 或 catch 块中执行 return、break 或 continue 语句或者使用 goto 跳出异常处理程序，也会执行 finally 语句块。这就在执行 try（可能还有 catch）语句块代码之后提供了一个可靠的清理方法。

如果没有指定 catch 语句块，那么 finally 语句块对于最终的资源清理也是非常有用的。如果将要编写的代码不能处理来自它所执行调用的异常，但又希望确保它使用的资源在栈中向上移动之前被正确地清理，就可以使用这种模式。下面的示例通过使用 using 关键字确保在 finally 语句块中正确地清理了 SqlConnection 和 SqlCommand。using 关键字用一个 try-finally 语句块包装了 using 语句的作用域。

```
public static int GetAuthorCount(string connectionString)
{
    SqlConnection sqlConn = null;
    SqlCommand sqlComm = null;

    using(sqlConn = new SqlConnection(connectionString))
    {
        using (sqlComm = new SqlCommand())
        {
            sqlComm.Connection = sqlConn;
            sqlComm.Parameters.Add("@pubName",
                SqlDbType.NChar).Value = "O''Reilly";
            sqlComm.CommandText = "SELECT COUNT(*) FROM Authors " +
                "WHERE Publisher=@pubName";

            sqlConn.Open();
            object authorCount = sqlComm.ExecuteScalar();
            return (int)authorCount;
        }
    }
}
```

在确定如何在应用程序或组件中组织异常处理之前，考虑执行以下操作。

- 在代码中较高层级的位置使用单个 try-catch 或 try-catch-finally 异常处理程序。可以将这种异常处理程序视 为粗粒度的。
- 调用栈中较低层级的代码应该包含 try-finally 异常处理程序。可以将这种异常处理程序视为细粒度的。

在异常发生后，细粒度的 try-finally 异常处理程序允许更好地控制清理工作。然后把异常冒泡到粗粒度的 try-catch 或 try-catch-finally 异常处理程序。这种技术允许一种更集中的异常处理模式，并且把必须编写用于处理异常的代码减至最少。

如果你知道代码将在单线程环境中运行，为了改进性能，应该处理可能引发异常的情况，而不是在引发异常后捕获它。如果代码将在多线程上运行，那么初始检查仍有可能成功，但是在能够采取的检查之后，对象值可能在操作之前在另一个线程中改变（也许变为 null）。

例如，在单线程环境中，如果一个方法有很大可能返回一个 null 值，那么在使用返回的值之前应该测试该值是否为 null，而不是使用 try-catch 语句块并允许引发 NullReferenceException。如果你认为 null 值是可能的，就要检查它。如果不应该发生这种情况，那么当它发生时就是一种异常条件，并且应该使用异常处理。为了说明这一点，下面给出了一个方法，它使用异常处理代码来处理 NullReferenceException。

```
public void SomeMethod()
{
    try
    {
        Stream s = GetAnyAvailableStream();
        Console.WriteLine("This stream has a length of " + s.Length);
    }
    catch (NullReferenceException)
    {
        // 此处处理空的流
    }
}
```

下面的方法代之以使用 if-else 条件语句来实现。

```
public void SomeMethod()
{
    Stream s = GetAnyAvailableStream();
    if (s != null)
    {
        Console.WriteLine("This stream has a length of " + s.Length);
    }
    else
    {
        // 此处处理空的流
    }
}
```

此外，你还应该确保以如下方式使用 finally 语句块关闭此流。

```
public void SomeMethod()
{
    Stream s = null;
    using(s = GetAnyAvailableStream())
    {
        if (s != null)
        {
            Console.WriteLine("This stream has a length of " + s.Length);
```

```
            }
            else
            {
                // 此处处理空的流
            }
        }
    }
```

finally 语句块包含将关闭流的方法调用，从而确保不会有数据丢失。

考虑引发异常，而不是返回错误代码。利用良好放置的异常处理代码，应该不必依靠返回错误代码（例如布尔值 true-false）的方法正确地处理错误，这可以使得代码更清晰。另一个好处是不必为错误代码查找任何值以理解该代码。

> 异常的最大优点是，当异常情况发生时，不能像对待错误代码那样仅仅忽略它。这有助于查找和修正错误。

要尽量引发特定的异常，而不是一般的异常。例如，引发一个 ArgumentNullException 而不是一个 ArgumentException，后者是前者的基类。引发一个 ArgumentException 只是告诉你方法的参数值有问题；引发一个 ArgumentNullException 则更确切地告诉你参数的真正问题是什么。另一个潜在的问题是，如果异常捕获程序正在寻找从引发的异常派生而来的更具体的类型，就可能不会捕获更一般的异常。

FCL 提供了几种异常类型，你将会发现在自己的代码中引发它们是非常有用的。下面列出了其中许多异常，并且定义了应该在何时、何处引发它们。

- 使用一个非适当状态的对象调用一个属性、索引器或方法时，引发一个 InvalidOperationException。（例如，对尚未初始化的对象调用一个索引器或者未按顺序地调用方法时。）
- 如果传入方法、属性或索引器中的参数无效，就引发 ArgumentException。ArgumentNullException、ArgumentOutOfRangeException 和 InvalidEnumArgumentException 是 ArgumentException 的三个子类，引发其中一个子类化的异常是更合适的，因为它们更清楚地指示了问题的根源。ArgumentNullException 指示传入的参数是 null，而这个参数在任何情况下都不能为 null。ArgimentOutOfRangeException 指示传入的参数在可接受的有效范围之外。这个异常主要用于数值。InvalidEnumArgumentException 指示传入的枚举值不在该枚举类型中。
- 当把一个无效的格式化参数作为参数传入一个方法中时，就引发一个 FormatException。这种技术主要用于重写 / 重载类似 ToString() 的接受格式化字符串的方法时，以及用在各种数字类型上的解析方法中。
- 当对一个已被处置的对象调用属性、索引或方法时，就引发一个 ObjectDisposedException。
- 从 SystemException 类派生而来的许多异常，例如 NullReferenceException、ExecutionEngineException、StackOverflowException、OutOfMemoryException 和 Index-

`OutOfRangeException`，只能由 CLR 引发，而不应该在代码中利用 `throw` 关键字显式引发它们。

.NET Framework 类库（FCL）中包含许多类，用于获得关于应用程序的诊断信息，以及它运行所在的环境。事实上，这样的类有很多，并且创建了一个命名空间 `System.Diagnostics` 来包含所有这些类。本章包含一些范例，利用调试 / 跟踪信息检测应用程序、获得进程信息、使用内置的事件日志，以及利用如性能计数器或者 Windows 事件跟踪（ETW）和 `EventSource` 等机制。应该指出的是，ETW 和 `EventSource` 正成为 .NET Framework 中首选的性能遥测机制。

默认情况下，只在调试构建时才打开调试支持（使用 `Debug` 类），而跟踪（使用 `Trace` 类）则在调试和发布构建时都会打开。这些默认情况允许交付使用 `Trace` 类的跟踪代码来检测的应用程序。交付编译有跟踪支持的代码，但在配置中关闭它，因此跟踪代码不会被调用（出于性能原因），除非它是服务器端应用程序（此类应用中，检测信息的价值要超出性能损失，并且在云中只能通过日志来了解程序行为）。如果在生产机器上发生在开发机器上不能重建的问题，就可以启用跟踪并允许把跟踪信息转储到一个文件中。然后可以检查该文件，以帮助查明真正的问题。

由于 `Debug` 类和 `Trace` 类都包含具有相同名称的相同成员，所以可以在代码中通过把 `Debug` 重命名为 `Trace`（反之亦然）来互换它们。本章中的大多数范例都使用 `Trace` 类；要修改这些范例以使用 `Dubug` 类代替，只需在代码中用 `Debug` 替换每个 `Trace` 实例即可。

5.1　知道何时捕获并重新引发异常

5.1.1　问题

你想确定何时捕获并重新引发一个异常是合适的。

5.1.2　解决方案

如果你有一个代码区域，当某个异常发生时你想在其中执行某种操作，但是不想采取任何操作实际地处理异常，那么捕获并重新引发该异常就是合适的。为了获取异常以便对它执行初始操作，可以建立一个 `catch` 语句块来捕获异常。然后，一旦执行了该操作，就从处理原始异常的 `catch` 语句块中重新引发异常。使用 `throw` 关键字后接一个分号来重新引发一个异常，代码如下所示。

```
try
{
    Console.WriteLine("In try");
    int z2 = 9999999;
    checked { z2 *= 999999999; }
}
catch (OverflowException oe)
{
    // 记录溢出异常发生的事实
```

```
        EventLog.WriteEntry("MyApplication", oe.Message, EventLogEntryType.Error);
        throw;
    }
```

此外创建了一个 EventLog 条目，用于记录发生的溢出异常。然后，通过 throw 语句将该异常冒泡到调用栈。

5.1.3　讨论

为异常建立一个 catch 语句块实质上是表达想对该异常情况执行某种操作。

> 如果你没有重新引发异常，或者创建一个新异常来包装原始的异常并引发它，那么就可以认为你已经处理了异常引发的情况，并且程序可以继续执行正常的操作。

通过选择重新引发异常，说明仍然有一个问题需要处理，并且依靠栈上面距离较远的代码来处理这种条件。如果你需要基于引发的异常执行某种操作并且需要在代码执行后允许异常继续存在，那么就可以用重新引发异常的机制来处理这种情况。如果这两个条件都不满足，就不要重新引发异常，只需处理它或者删除 catch 语句块。

> 要记得引发异常的代价很高。不要尝试不必要地引发和重新引发异常，因为这可能使应用程序陷入困境。

在重新引发异常时，使用 throw;，而不要使用 throw ex;，因为 throw; 将保留异常的原始调用栈。使用带有 catch 参数的 throw 将把调用栈重设到那个位置，并且关于错误的信息也将会丢失。可能在某些情况下你想要更改调用堆栈（例如要隐藏应用程序执行敏感操作部分的内部细节），但整体来说，给自己最好的机会去调试而不要截断调用堆栈。

5.2　处理通过反射调用的方法引发的异常

5.2.1　问题

使用反射可以调用一个会生成异常的方法。你希望获得真正的异常对象及其信息，以便诊断和修正问题。

5.2.2　解决方案

可以通过 MethodInfo.Invoke 引发的 TargetInvocationException 的 InnerException 属性获得真正的异常及其信息。

5.2.3 讨论

例 5-1 处理了在通过反射调用的方法内发生的一个异常。Reflect 类包含一个 ReflectionException 方法，它使用反射类调用静态方法 TestInvoke。

例 5-1：获得通过反射调用的方法所引发异常的信息

```
using System;
using System.Reflection;

public static class Reflect
{
    public static void ReflectionException()
    {
        Type reflectedClass = typeof(DebuggingAndExceptionHandling);

        try
        {
            MethodInfo methodToInvoke = reflectedClass.GetMethod("TestInvoke");
            methodToInvoke?.Invoke(null, null);
        }
        catch(Exception e)
        {
            Console.WriteLine(e.ToShortDisplayString());
        }
    }

    public static void TestInvoke()
    {
        throw (new Exception("Thrown from invoked method."));
    }
}
```

这段代码显示以下文本。

```
Message: Exception has been thrown by the target of an invocation.
Type: System.Reflection.TargetInvocationException
Source: mscorlib
TargetSite: System.Object InvokeMethod(System.Object, System.Object[], System.Si
gnature, Boolean)
**** INNEREXCEPTION START ****
Message: Thrown from invoked method.
Type: System.Exception
Source: CSharpRecipes
TargetSite: Void TestInvoke()
**** INNEREXCEPTION END ****
```

当调用 methodToInvoke?.Invoke 方法时，会调用 TestInvoke 方法，它引发一个异常。紧跟着 methodToInvoke 的问号是一个 null 条件运算符，用来处理无法获得 MethodInfo 时其值为 null 的情况。通过这种方式，就不需要在调用时编写 null 值的检查语句。外部异常是 TargetInvocationException；这是一个通用异常，当通过反射调用的方法引发一个异常时将引发该异常。CLR 会自动将调用的方法引发的原始异常包装到 TargetInvocationException 对象的 InnerException 属性中。在这种情况下，由调用的方法

引发的异常是 System.Exception 类型。该异常显示在以文本 ****INNEREXCEPTIONSTART**** 开头的区域后面。

为了显示异常信息，可调用 ToShortDisplayString 方法，代码如下所示。

```
Console.WriteLine(e.ToShortDisplayString());
```

Exception 的 ToShortDisplayString 扩展方法使用 StringBuilder 创建关于异常以及所有内部异常的信息串。WriteExceptionShortDetail 方法利用异常数据的特定部分填充 StringBuilder。为了获取内部异常，可使用 GetNestedExceptionList 扩展方法，代码如下所示。

```
public static string ToShortDisplayString(this Exception ex)
{
    StringBuilder displayText = new StringBuilder();
    WriteExceptionShortDetail(displayText, ex);
    foreach(Exception inner in ex.GetNestedExceptionList())
    {
       displayText.AppendFormat("**** INNEREXCEPTION START ****{0}",
           Environment.NewLine);
       WriteExceptionShortDetail(displayText, inner);
       displayText.AppendFormat("**** INNEREXCEPTION END ****{0}{0}",
           Environment.NewLine);
    }
    return displayText.ToString();
}

public static IEnumerable<Exception> GetNestedExceptionList(
         this Exception exception)
{
    Exception current = exception;
    do
    {
        current = current.InnerException;
        if (current != null)
            yield return current;
    }
    while (current != null);
}

public static void WriteExceptionShortDetail(StringBuilder builder, Exception ex)
{
    builder.AppendFormat("Message: {0}{1}", ex.Message, Environment.NewLine);
    builder.AppendFormat("Type: {0}{1}";, ex.GetType(), Environment.NewLine);
    builder.AppendFormat("Source: {0}{1}", ex.Source, Environment.NewLine);
    builder.AppendFormat("TargetSite: {0}{1}", ex.TargetSite,
        Environment.NewLine);
}
```

5.2.4 参考

MSDN 文档中的“Type 类”“Null 条件运算符”和“MethodInfo 类”主题。

5.3 创建新的异常类型

5.3.1 问题

.NET Framework 中的所有内置异常都没有提供你需要引发的异常的实现细节。你需要创建自己的异常类，它可以与你的应用程序以及其他应用程序无缝地协同工作。无论何时应用程序接收到这个新异常，它都可以通知用户在特定的组件中发生了特定的错误。这种报告将极大地减少调试问题所需的时间。

5.3.2 解决方案

创建自己的异常类。为了说明这一点，让我们创建一个自定义的异常类 RemoteComponentFxception，它将通知客户端应用程序在远程服务器程序集中发生了错误。

5.3.3 讨论

异常层次结构开始于 Exception 类，从该类派生出两个类：ApplicationException 和 SystemException。SystemException 类及其派生的任何类是为 FCL 开发人员预留的。大多数常见的异常（如 NullReferenceException 或 OverflowException）都是从 SystemException 派生而来的。FCL 开发人员为使用 .NET 语言的其他开发人员创建了 App1icationException 类，用于派生出他们自己的异常。这种划分让我们可以清楚地区分用户定义的异常与内置的系统异常。不过，Microsoft 现在建议直接从 Exception，而不是从 ApplicationException 派生异常。没有什么会能主动阻止你从 SystemException 或 ApplicationException 派生一个类。但是最好保持一致，并且遵守如下约定：总是从 Exception 类派生用户定义的异常。

在确定异常的名称时，应该遵守异常的命名约定。该约定非常简单：决定异常的名称，并在该名称末尾添加单词 Exception（例如，使用 UnknownException 作为异常的名称，而不是仅使用 Unknown）。

每个用户定义的异常都应该包括至少三个构造函数，下面将逐一介绍。这不是一种要求，但它会使你的异常类的工作方式类似于 FCL 中其他所有异常类，并且可以尽量缩短其他开发人员使用你的新异常的学习曲线。

- *默认构造函数*

  这个构造函数不带参数，并且只调用基类的默认构造函数。

- *带有一个参数（接受消息字符串）的构造函数*

  这个消息字符串将会重写该异常的 Message 字段的默认内容。像默认构造函数一样，这个构造函数也会调用基类的构造函数，它也接受一个消息字符串作为它的唯一参数。

- *接受消息字符串和内部异常作为参数的构造函数*

  将 innerException 参数中包含的对象添加到这个异常对象的 InnerException 属性中。像另外两个构造函数一样，这个构造函数也会调用基类具有相同签名的构造函数。

应该创建一些字段及其访问器以保存异常特有的数据。因为该异常是因为在远程服务器程序集中发生错误而引发的，所以将添加一个私有字段来包含服务器或服务的名称。此外，还将添加一个公共的只读属性用于访问这个字段。因为要添加这个新字段，所以应该添加两个构造函数，它们接受一个额外的参数，用于设置 serverName 字段的值。

如果必要，可重写其行为被自定义异常类继承的任何基类的成员。例如，因为添加了一个新字段，所以需要确定是否要为这个异常把它添加到 Message 字段的默认内容中。如果是这样，就必须重写 Message 属性，代码如下所示。

```
public override string Message => $"{base.Message}{Environment.NewLine}" +
            $"The server ({this.ServerName ?? "Unknown"})" +
           "has encountered an error.";
```

注意在第一行显示基类中的 Message 属性，并在下一行显示额外的文本。这种组织方式考虑到用户可能会使用接受消息字符串作为参数的重载构造函数之一来修改将出现在 Message 属性中的消息。

你的异常对象应该可以序列化和反序列化。这涉及执行下面两个额外的步骤。

(1) 将 Serializable 特性添加到类定义中。这个特性指定类可以序列化和反序列化。如果类中没有这个特性，并且尝试序列化该类，就会引发一个 SerializationException。
(2) 如果想控制如何执行序列化和反序列化，类就应该实现 ISerializable 接口，并且应该为它的单个成员 GetObjectData 提供一种实现。这里实现它是因为基类实现了它，这意味着如果想序列化添加的字段（例如 serverName），就必须重新实现它。

```
// 用于在序列化时捕获额外字段的信息
public override void GetObjectData(SerializationInfo exceptionInfo,
                                   StreamingContext exceptionContext)
{
    base.GetObjectData(exceptionInfo, exceptionContext);
    exceptionInfo.AddValue("ServerName", this.ServerName);
}
```

此外，还需要一个新的构造函数重写，它接受信息以反序列化该对象，代码如下所示。

```
// 序列化构造函数
protected RemoteComponentException(SerializationInfo exceptionInfo,
        StreamingContext exceptionContext)
        : base(exceptionInfo, exceptionContext)
{
    this.serverName = exceptionInfo.GetString("ServerName");
}
```

即使不要求这样做，也应该使所有用户定义的异常类都是可序列化和反序列化的。这样，就可以通过远程方式以及在应用程序域边界上正确地传播此异常。

如果要在非托管代码（比如 COM 对象）中捕获这个异常，还可以为该异常设置 HRESULT 值。在非托管代码中捕获的异常将变成一个 HRESULT 值。如果异常没有改变 HRESULT 值，它就默认为基类异常的 HRESULT，就继承自 ApplicationException 的用户定义的异常对象

来说，它是 COR_E_APPLICATION（0x80131600）。为了改变默认的 HRESULT 值，只需在构造函数中设置这个字段的值。下面的代码演示了这种技术。

```
public class RemoteComponentException : Exception
{
    public RemoteComponentException() : base()
    {
         HResult = 0x80040321;
    }

    public RemoteComponentException(string message) :
        base(message)
    {
        HResult = 0x80040321;
    }

    public RemoteComponentException(string message, Exception innerException)
        : base(message, innerException)
    {
        HResult = 0x80040321;
    }
}
```

现在，COM 对象将看到的 HRESULT 的值为 0x80040321。

> 重写 Message 属性以将任何新字段纳入到异常消息文本中通常是一个好主意。始终记住，要包括基类的消息文本以及添加到该属性中的任何额外文本。

此时，RemoteComponentException 类提供了创建一个完整的用户定义的异常类所需的一切内容。

最后指出一点，把所有用户定义的异常放在一个单独的程序集中是一个好主意。这样可以更容易地在其他应用程序中重用这些异常。更重要的是，其他应用程序域和远程执行代码还可以正确地引发和处理这些异常，而不管它们是在哪里引发的。应该用强名称签署保存这些异常的程序集，并把它添加到*全局程序集缓存*（global assembly cache，GAC）中，以便任何使用或处理这些异常的代码可以找到定义它们的程序集。参见 11.7 节，了解关于如何执行该任务的更多信息。

如果你确信将要定义的异常永远不会在程序集外面引发或处理，那么就可以把异常定义留在那里。但是，如果出于某种原因需要在程序集外面处理引发的异常，那么最终捕获它的代码将不能解析它。

RemoteComponentException 类的完整源代码如例 5-2 所示。

例 5-2：RemoteComponentException 类

```
using System;
using System.IO;
using System.Runtime.Serialization;
```

```
using System.Runtime.Serialization.Formatters.Binary;
using System.Security.Permissions;

[Serializable]
public class RemoteComponentException : Exception, ISerializable
{
    #region Constructors
    // 普通的异常构造函数
    public RemoteComponentException() : base()
    {
    }

    public RemoteComponentException(string message) : base(message)
    {
    }

    public RemoteComponentException(string message, Exception innerException)
        : base(message, innerException)
    {
    }

    // 接受新的serverName参数的构造函数
    public RemoteComponentException(string message, string serverName) :
        base(message)
    {
        this.ServerName = serverName;
    }

    public RemoteComponentException(string message,
                 Exception innerException, string serverName)
        : base(message, innerException)
    {
        this.ServerName = serverName;
    }

    // 序列化构造函数
    protected RemoteComponentException(SerializationInfo exceptionInfo,
        StreamingContext exceptionContext)
        : base(exceptionInfo, exceptionContext)
    {
        this.ServerName = exceptionInfo.GetString("ServerName");
    }
    #endregion // Constructors

    #region Properties
    // 服务器名称只读属性
    public string ServerName { get; }

    public override string Message => $"{base.Message}{Environment.NewLine}" +
                $"The server ({this.ServerName ?? "Unknown"})" +
                "has encountered an error.";
    #endregion // Properties

    #region Overridden methods
    // ToString方法
```

```
    public override string ToString() =>
         "An error has occurred in a server component of this client." +
            $"{Environment.NewLine}Server Name: " +
            $"{this.ServerName}{Environment.NewLine}" +
            $"{this.ToFullDisplayString()}";


    // 用于在序列化时捕获额外字段的信息
    [SecurityPermission(SecurityAction.LinkDemand, Flags =
         SecurityPermissionFlag.SerializationFormatter)]
    public override void GetObjectData(SerializationInfo info,
        StreamingContext context)
    {
        base.GetObjectData(info, context);
        info.AddValue("ServerName", this.ServerName);
    }
    #endregion // Overridden methods

    public string ToBaseString() => (base.ToString());
}
```

重写的 ToString 方法中执行的 ToFullDisplayString 是 Exception 类的一个扩展方法，另一个扩展方法 GetNestedExceptionList 用于获得异常列表，而 WriteExceptionDetail 扩展方法用于处理每个 Exception 的细节。

```
public static string ToFullDisplayString(this Exception ex)
{
    StringBuilder displayText = new StringBuilder();
    WriteExceptionDetail(displayText, ex);
    foreach (Exception inner in ex.GetNestedExceptionList())
    {
        displayText.AppendFormat("**** INNEREXCEPTION START ****{0}",
            Environment.NewLine);
        WriteExceptionDetail(displayText, inner);
        displayText.AppendFormat("**** INNEREXCEPTION END ****{0}{0}",
            Environment.NewLine);
    }
    return displayText.ToString();
}

public static IEnumerable<Exception> GetNestedExceptionList(
    this Exception exception)
{
    Exception current = exception;
    do
    {
        current = current.InnerException;
        if (current != null)
            yield return current;
    }
    while (current != null);
}

public static void WriteExceptionDetail(StringBuilder builder, Exception ex)
{
```

```
    builder.AppendFormat("Message: {0}{1}", ex.Message, Environment.NewLine);
    builder.AppendFormat("Type: {0}{1}", ex.GetType(), Environment.NewLine);
    builder.AppendFormat("HelpLink: {0}{1}", ex.HelpLink, Environment.NewLine);
    builder.AppendFormat("Source: {0}{1}", ex.Source, Environment.NewLine);
    builder.AppendFormat("TargetSite: {0}{1}", ex.TargetSite,
        Environment.NewLine);
    builder.AppendFormat("Data:{0}", Environment.NewLine);
    foreach (DictionaryEntry de in ex.Data)
    {
        builder.AppendFormat("\t{0} : {1}{2}",
            de.Key, de.Value, Environment.NewLine);
    }
    builder.AppendFormat("StackTrace: {0}{1}", ex.StackTrace,
        Environment.NewLine);
}
```

例 5-3 中显示了用于测试 RemoteComponentException 类的部分代码清单。

例 5-3：测试 RemoteComponentException 类

```
public void TestSpecializedException()
{
    // 生成用于测试RemoteComponentException的innerException的内部异常
    Exception inner = new Exception("The inner Exception");

    RemoteComponentException se1 = new RemoteComponentException ();
    RemoteComponentException se2 =
      new RemoteComponentException ("A Test Message for se2");
    RemoteComponentException se3 =
      new RemoteComponentException ("A Test Message for se3", inner);
    RemoteComponentException se4 =
      new RemoteComponentException ("A Test Message for se4",
                                    "MyServer");
    RemoteComponentException se5 =
      new RemoteComponentException ("A Test Message for se5", inner,
                                    "MyServer");

    // 测试重写的Message属性
    Console.WriteLine(Environment.NewLine +
      "TEST -OVERRIDDEN- MESSAGE PROPERTY");
    Console.WriteLine("se1.Message == " + se1.Message);
    Console.WriteLine("se2.Message == " + se2.Message);
    Console.WriteLine("se3.Message == " + se3.Message);
    Console.WriteLine("se4.Message == " + se4.Message);
    Console.WriteLine("se5.Message == " + se5.Message);

    // 测试重写的ToString方法
    Console.WriteLine(Environment.NewLine +
      "TEST -OVERRIDDEN- TOSTRING METHOD");
    Console.WriteLine("se1.ToString() == " + se1.ToString());
    Console.WriteLine("se2.ToString() == " + se2.ToString());
    Console.WriteLine("se3.ToString() == " + se3.ToString());
    Console.WriteLine("se4.ToString() == " + se4.ToString());
    Console.WriteLine("se5.ToString() == " + se5.ToString());
    Console.WriteLine(Environment.NewLine + "END TEST" + Environment.NewLine);
}
```

例 5-4 中展示了例 5-3 的输出。

例 5-4：`RemoteComponentException` 类显示的输出

```
TEST -OVERRIDDEN- MESSAGE PROPERTY
se1.Message == Exception of type 'CSharpRecipes.ExceptionHandling+RemoteComponen
tException' was thrown.
A server with an unknown name has encountered an error.
se2.Message == A Test Message for se2
A server with an unknown name has encountered an error.
se3.Message == A Test Message for se3
A server with an unknown name has encountered an error.
se4.Message == A Test Message for se4
The server (MyServer) has encountered an error.
se5.Message == A Test Message for se5
The server (MyServer) has encountered an error.

TEST -OVERRIDDEN- TOSTRING METHOD
se1.ToString() == An error has occurred in a server component of this client.
Server Name:
Message: Exception of type 'CSharpRecipes.ExceptionHandling+RemoteComponentExcep
tion' was thrown.
A server with an unknown name has encountered an error.

Type: CSharpRecipes.ExceptionHandling+RemoteComponentException
HelpLink:
Source:
TargetSite:
Data:
StackTrace:

se2.ToString() == An error has occurred in a server component of this client.
Server Name:
Message: A Test Message for se2
A server with an unknown name has encountered an error.
Type: CSharpRecipes.ExceptionHandling+RemoteComponentException
HelpLink:
Source:
TargetSite:
Data:
StackTrace:

se3.ToString() == An error has occurred in a server component of this client.
Server Name:
Message: A Test Message for se3
A server with an unknown name has encountered an error.
Type: CSharpRecipes.ExceptionHandling+RemoteComponentException
HelpLink:
Source:
TargetSite:
Data:
StackTrace:
**** INNEREXCEPTION START ****
Message: The Inner Exception
Type: System.Exception
```

```
HelpLink:
Source:
TargetSite:
Data:
StackTrace:
**** INNEREXCEPTION END ****

se4.ToString() == An error has occurred in a server component of this client.
Server Name: MyServer
Message: A Test Message for se4
The server (MyServer) has encountered an error.
Type: CSharpRecipes.ExceptionHandling+RemoteComponentException
HelpLink:
Source:
TargetSite:
Data:
StackTrace:

se5.ToString() == An error has occurred in a server component of this client.
Server Name: MyServer

Message: A Test Message for se5
The server (MyServer) has encountered an error.
Type: CSharpRecipes.ExceptionHandling+RemoteComponentException
HelpLink:
Source:
TargetSite:
Data:
StackTrace:
**** INNEREXCEPTION START ****
Message: The Inner Exception
Type: System.Exception
HelpLink:
Source:
TargetSite:
Data:
StackTrace:
**** INNEREXCEPTION END ****
END TEST
```

5.3.4 参考

范例 11.7（即 11.7 节）；MSDN 文档中的“使用用户定义的异常”和“Exception 类”主题。

5.4 在首次异常上中断

5.4.1 问题

你需要修正一段引发异常的代码中的问题。不幸的是，异常处理程序正在捕获异常，你很难查明在何时、何处引发了异常。

如果你需要在第一次引发异常的位置单步调试代码，那么在应用程序有机会处理异常之前强制应用程序在异常上中断是非常有用的。如果应用程序引发该异常但未处理它，调试器将进行干预，并在引发未处理异常的代码行上中断。在这种情况下，可以查看引发异常的环境。不过，如果在引发异常时异常处理程序是活动的，异常处理程序将会处理异常并继续运行，阻止你看到引发异常的那个位置的环境。这是所有异常的默认行为。

5.4.2 解决方案

在 Visual Studio 2015 内选择 Debug → Exceptions 或者使用 Ctrl-Alt-E 组合键，将会显示 Exceptions Settings 工具窗口（参见图 5-1）。从树视图中选择你想修改的异常，然后单击树视图中的复选框。单击 OK 按钮，然后运行应用程序。无论应用程序何时引发一个 `System.ArgumentOutOfRangeException` 异常，在应用程序有机会处理该异常之前，调试器将在该代码行上中断。

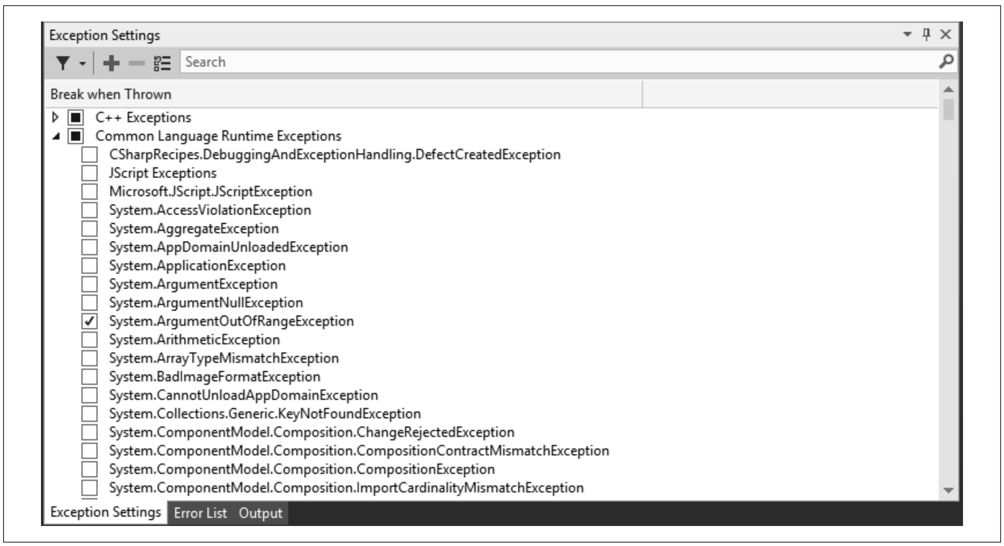


图 5-1：Exceptions Settings 工具窗口

使用 Exceptions Settings 工具窗口，可以定位你想改变其默认行为的特定异常或异常集。该对话框具有三个主要区域。第一个是 TreeView 控件，它包含分类的异常列表。使用这个 TreeView，可以选择一个或多个你希望修改其行为的异常或异常组。

这个对话框的下一个区域是 TreeView 旁列表中的 Thrown 列。该列包含与每个异常对应的复选框，在第一次引发相应的异常类型时它允许调试器中断。在这个阶段，异常被视为首次异常。如果在 Thrown 列中选中某个复选框，当引发在 TreeView 控件中所选的首次异常类型时，将强制调试器进行干预。如果不选中复选框，就允许应用程序尝试处理第一次机会的异常。

还可以单击窗口左上方的 Filter 图标，将异常视图筛选为只显示你选中在首次异常时中断的所有异常，如图 5-2 所示。

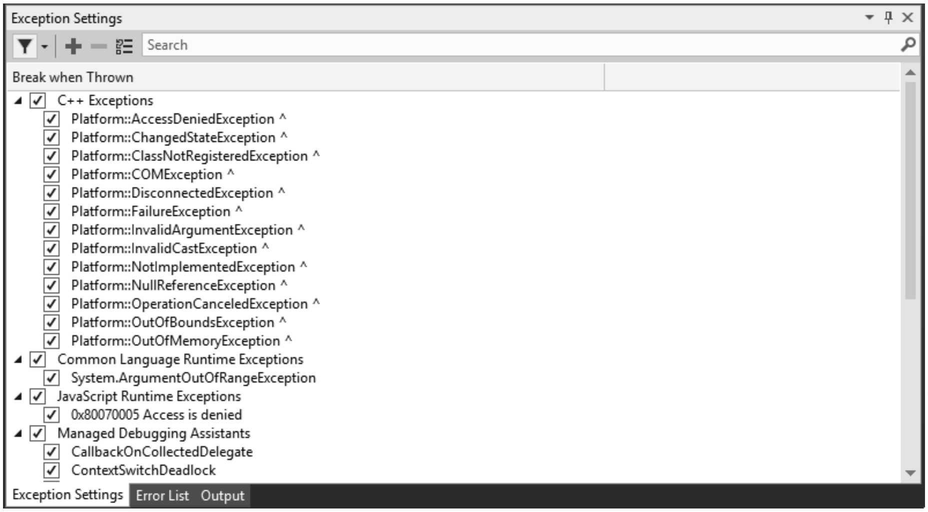


图 5-2：筛选过的 Exceptions Settings 工具窗口

Exceptions Settings 工具窗口上部还提供了一个搜索条，允许你搜索窗口中的异常。如果在窗口中输入 `argumentnullexception`，会看到选择缩窄为只包含匹配文本的异常，如图 5-3 所示。

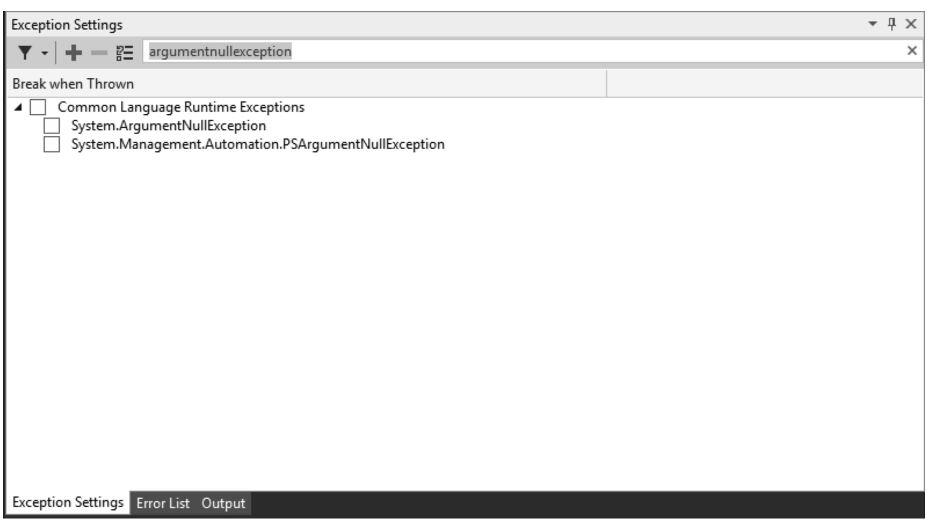


图 5-3：Exceptions Settings 工具窗口搜索

要把用户定义的异常添加到 Exception Settings 中，可单击 Add 按钮。将显示如图 5-4 所示的对话框。



图 5-4：把用户定义的异常添加到 Exception Settings

单击 Yes 使用原始的 Exceptions 对话框，如图 5-5 所示。

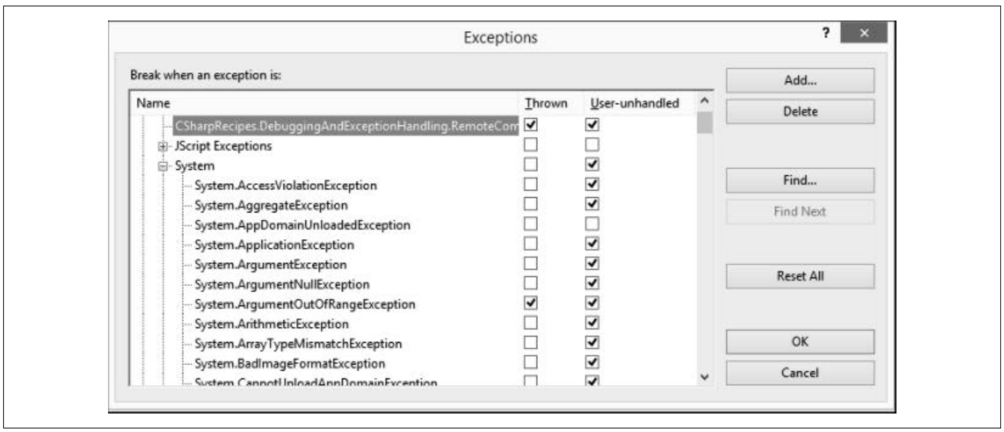


图 5-5：Exceptions 对话框

这个对话框包含两个有用的按钮，Find 和 Find Next。它们允许查找一个异常而不用深入检查 TreeView 控件，并且可以根据自己的需要进行查找。此外，还有另外三个按钮（Reset All、Add 和 Delete）分别用于复位到原始状态，以及添加和删除用户定义的异常。

例如，你可以创建自己的异常，就像在 5.3 节中所做的那样，并且把该异常添加到列表中。必须把任何像这样的托管异常添加到名为 Common Language Runtime Exceptions 的 TreeView 节点下。这一设置告诉调试器这是一个托管异常，并且以托管异常的方式对它进行处理。图 5-6 显示了添加自定义的异常。

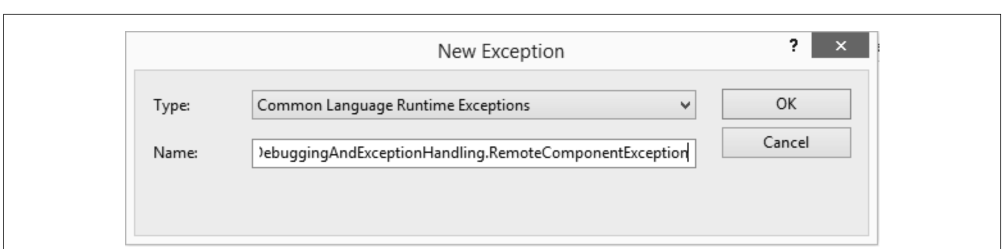


图 5-6：将一个用户定义的异常添加到 Exceptions 对话框

在该对话框的 Name 字段中输入异常的名称，包括其准确的类名以及完整的命名空间作用域。不要在该名称中追加任何其他信息，比如它驻留的命名空间以及它嵌套的类名。如果这样做，在引发异常时调试器将无法看到它。单击 OK 按钮，将异常置于 TreeView 中的 Common Language Runtime Exceptions 节点下。在添加用户定义的异常之后，Exceptions 对话框看起来如图 5-7 所示。

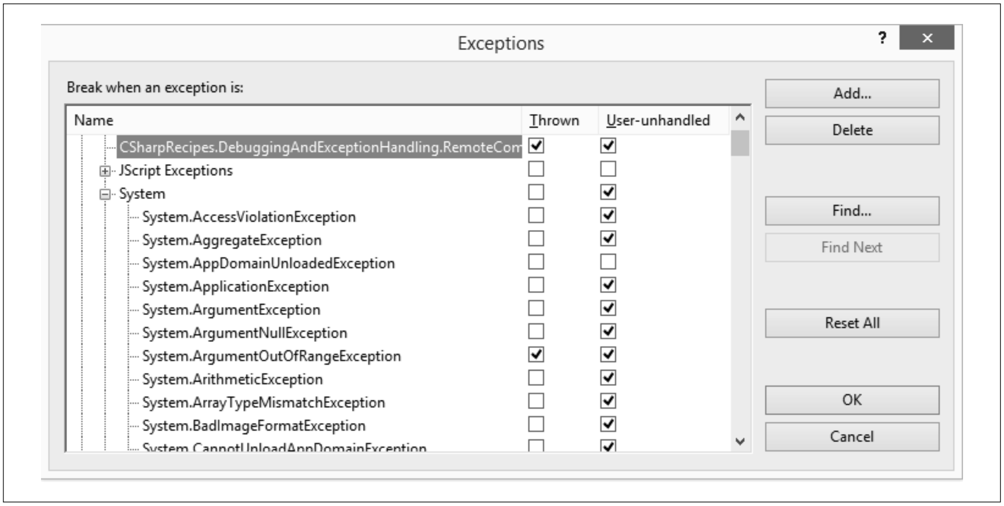


图 5-7：将用户定义的异常添加到 TreeView 中之后的 Exceptions 对话框

Delete 按钮用于删除添加到 TreeView 中的所有已选中用户定义的异常。Reset All 按钮用于删除添加到 TreeView 中的所有用户定义的异常。选中 Thrown 列中的复选框，当引发相应的异常类型时将会使调试器停下。

还有另外一个设置可能会影响异常调试：Just My Code（详见图 5-8）。应该关闭这个设置，以便在调试时可以最清晰地看到应用程序中实际发生的事情；当启用此设置时，将无法看到代码调用的框架代码的相关行为。能够看到你的代码调入了框架的何处，以及何处离开框架代码是非常有教育意义的，能够帮助你更好地理解你正在调试的问题。该设置位于 Visual Studio 2015 中的 Tools\Options\Debugging\General 之下。

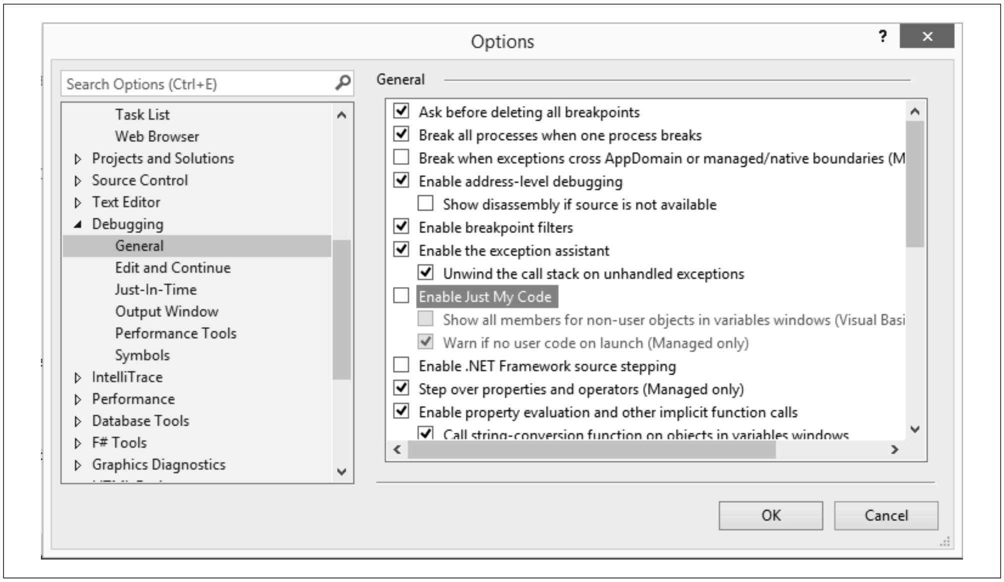


图 5-8：禁用 Just My Code 设置

5.4.3 参考

MSDN 文档中的“异常处理（调试）”主题。

5.5 处理从异步委托中引发的异常

5.5.1 问题

当异步使用委托时，如果该委托引发任何异常，你都希望得到通知。

5.5.2 解决方案

在 try-catch 语句块中包装委托的 EndInvoke 方法，代码如下所示。

```
using System;
using System.Threading;

public class AsyncAction
{
    public void PollAsyncDelegate()
    {
        // 创建一个异步委托以调用Method1,并调用委托的
        // BeginInvoke方法
        AsyncInvoke MI = new AsyncInvoke(TestAsyncInvoke.Method1);
        IAsyncResult AR = MI.BeginInvoke(null, null);
```

```
            // 轮循直到异步委托完成
            while (!AR.IsCompleted)
            {
               System.Threading.Thread.Sleep(100);
               Console.Write('.');
            }
            Console.WriteLine("Finished Polling");
            // 调用异步委托的EndInvoke方法
            try
            {
                int RetVal = MI.EndInvoke(AR);
                Console.WriteLine("RetVal (Polling): " + RetVal);
            }
            catch (Exception e)
            {
                Console.WriteLine(e.ToString());
            }
        }
    }
```

下面的代码定义了 AsyncInvoke 委托和异步调用的静态方法 TestAsyncInvoke.Method1。

```
    public delegate int AsyncInvoke();

    public class TestAsyncInvoke
    {
        public static int Method1()
        {
            throw (new Exception("Method1")); // 模拟一个引发的异常
        }
    }
```

5.5.3 讨论

如果 PollAsyncDelegate 方法中的代码不包含对委托的 EndInvoke 方法的调用，在 Method1 中引发的异常将会被简单地丢弃并且永远不会被捕获，或者如果应用程序绑定了顶级异常处理程序（参见范例 5.2、范例 5.7 和范例 5.8，即 5.2 节、5.7 节和 5.8 节），就会捕获异常。如果调用 EndInvoke，那么异常将在调用 EndInvoke 时触发并且可以在那里被捕获。这种行为是有意这样设计的，对于线程发生的所有未处理的异常，线程会立即返回到线程池，并且会丢失异常。

如果通过委托异步调用的方法引发一个异常，捕获该异常的唯一方式是包括对委托的 EndInvoke 方法的调用，并把该调用包装在一个异常处理程序中。必须调用 EndInvoke 方法获取异常委托的结果，事实上，即使没有任何结果，也必须调用 EndInvoke 方法。可以通过委托的返回值或者任何 ref 或 out 参数获得这些结果。

5.5.4 参考

有关在应用程序中绑定顶级异常处理程序的信息，参见范例 5.2、范例 5.7 和范例 5.8，即 5.2 节、5.7 节和 5.8 节。

5.6 利用Exception.Data为异常提供所需的额外信息

5.6.1 问题

你想随异常一起发送一些额外的信息。

5.6.2 解决方案

使用 System.Exception 对象的 Data 属性，存储与异常有关的键值对信息。

例如，假定有一个 System.ArgumentException 将要从某个代码区域中引发，而且你想让其中包括底层的原因以及它需要花费的时间长度。可以在 Exception.Data 属性中添加两个键值对，通过在索引器中指定键然后赋值。

在下面的示例中，异常对象的 Data 属性使用 "Cause" 和 "Length" 作为它的键。一旦在 Data 集合中建立了这些数据项，就可以引发异常并捕获它，并且可以在后续的 catch 块中添加更多数据，以便满足与将要遍历的异常一样多层次的异常处理。

```
try
{
    try
    {
        try
        {
            try
            {
                ArgumentException irritable =
                    new ArgumentException("I'm irritable!");
                irritable.Data["Cause"]="Computer crashed";
                irritable.Data["Length"]=10;
                throw irritable;
            }
            catch (Exception e)
            {
                // 确定是否可处理
                if(e.Data.Contains("Cause"))
                    e.Data["Cause"]="Fixed computer"
                throw;
            }
        }
        catch (Exception e)
        {
            e.Data["Comment"]="Always grumpy you are";
            throw;
        }
    }
    catch (Exception e)
    {
```

```
            e.Data["Reassurance"]="Error Handled";
            throw;
        }
    }
```

最后的 catch 语句块可以迭代 Exception.Data 集合并显示所有自初始异常被引发后在 Data 集合中收集的支持数据。

```
    catch (Exception e)
    {
        Console.WriteLine("Exception supporting data:");
        foreach(DictionaryEntry de in e.Data)
        {
            Console.WriteLine("\t{0} : {1}",de.Key,de.Value);
        }
    }
```

5.6.3 讨论

Exception.Data 是一个支持 IDictionary 接口的对象。使用该对象可以执行以下操作。

- 添加和删除名称 / 值对。
- 清除内容。
- 查找集合，看看它是否包含某个键。
- 获得一个 IDictionaryEnumerator 用于遍历集合项目。
- 使用键索引到集合。
- 单独访问所有键和所有值的 ICollection。

> 添加到 Exception.Data 的数据项需要是 Serializable，否则在添加到集合时将引发 ArgumentException。如果你要把一个类添加到 Exception.Data，请将其标记为 Serializable 并且确保它是可序列化的。

```
    public void TestExceptionDataSerializable()
    {
        Exception badMonkey =
            new Exception("You are a bad monkey!");
        try
        {
            badMonkey.Data["Details"] = new Monkey();
        }
        catch (ArgumentException aex)
        {
            Console.WriteLine(aex.Message);
        }
    }

    //[Serializable]  // 取消注释以使其可序列化
    public class Monkey
    {
        public string Name { get; } = "George";
    }
```

能够把特定于代码的数据附加到系统异常上是一件非常方便的事情，因为在发生错误时它允许更完整地查看代码中所发生的事情。它可以给可怜的人（可能是你自己）提供更多的信息，帮助他们首先查明为什么会引发异常，提供修正它的更好机会。在引发异常时给你自己和你的团队提供一点额外的信息；你不会后悔这样做的。

5.6.4　参考

MSDN 文档中的“Exception.Data 属性”主题。

5.7　在WinForms应用程序中处理未经处理的异常

5.7.1　问题

你有一个基于 WinForms 的应用程序，你想捕获并记录其中任何线程上的任何未经处理的异常。

5.7.2　解决方案

你需要为 System.Windows.Forms.Application.ThreadException 事件和 System.AppDomain.UnhandledException 事件挂接处理程序。这两个事件都需要挂接，因为 Framework 中的 WinForms 支持本身会做许多异常捕获工作。它会公开 System.Windows.Forms.Application.ThreadException 事件，允许获取 WinForms 及其事件的 UI 线程上发生的任何未经处理的异常。不要被其名称所欺骗，System.Windows.Forms.Application.ThreadException 事件处理程序不会捕获程序构造的工作者线程或者来自 ThreadPool 线程池中工作者线程上未经处理的异常。为了捕获 WinForms 应用程序中发生未捕获异常的所有可能路径，需要为 System.AppDomain.UnhandledException 事件挂接一个处理程序（System.Windows.Forms.Application.ThreadException 事件会捕获 UI 线程中的异常）。

为了挂接必要的事件处理程序以捕获 WinForms 应用程序中所有未经处理的异常，可在应用程序中的 Main 函数中添加以下代码。

```
static void Main()
{
    // 添加事件处理程序以捕获主UI线程中发生的所有异常
    Application.ThreadException +=
        new ThreadExceptionEventHandler(OnThreadException);

    // 为应用程序域中除UI线程之外的其他所有线程
    // 添加事件处理程序
    AppDomain.CurrentDomain.UnhandledException +=
        new UnhandledExceptionEventHandler(CurrentDomain_UnhandledException);

    Application.EnableVisualStyles();
    Application.Run(new Form1());
}
```

使用 AppDomain.CurrentDomain 的属性将 System.AppDomain.UnhandledException 事件处理程序挂接到当前的 AppDomain，这一属性允许你访问当前的 AppDomain。通过 Application.ThreadException 属性访问应用程序的 ThreadException 处理程序。

在 CurrentDomain_UnhandledException 和 OnThreadException 处理程序方法中建立事件处理程序代码。有关 UnhandledExceptionEventHandler 的更多信息，请参见范例 5.8（即 5.8 节）。ThreadExceptionEventHandler 接收发送方对象和一个 ThreadExceptionEventArgs 对象。ThreadExceptionEventArgs 包含一个 Exception 属性，它包含来自 WinForms UI 线程的未经处理的异常。

```
// 处理其他所有线程的异常事件
static void CurrentDomain_UnhandledException(object sender,
                            UnhandledExceptionEventArgs e)
{
    // 仅是显示异常详细信息
    MessageBox.Show("CurrentDomain_UnhandledException: " +
            e.ExceptionObject.ToString());
}

// 处理来自UI线程的异常事件
static void OnThreadException(object sender, ThreadExceptionEventArgs t)
{
    // 仅是显示异常详细信息
    MessageBox.Show("OnThreadException: " + t.Exception.ToString());
}
```

5.7.3 讨论

异常是在 .NET 中传达错误的主要方式，因此在构建应用程序时，必须包含未经处理异常的最后一道防线。未经处理的异常将会使程序崩溃（即使在 .NET 中看起来稍好一点）；你可不希望给自己的客户留下这种印象。为所有未经处理的异常挂接一个事件处理程序是一种良好的做法。AppDomain.UnhandledException 事件就是一个非常好的选择，但是不得不处理另外一个事件也不是世界末日。在为 AppDomain.UnhandledException 和 Application.ThreadException 编码事件处理程序时，可以轻松地调用单个处理程序，把异常信息写到事件日志、调试流或自定义的跟踪日志中，甚至给你发送一封带有该信息的电子邮件。其可能性受到的唯一限制是，你希望采用什么方式来处理任何程序可能发生的错误。

5.7.4 参考

MSDN 文档中的“ThreadExceptionEventHandler 委托”和“UnhandledExceptionEventHandler 委托”主题。

5.8 在WPF应用程序中处理未经处理的异常

5.8.1 问题

你有一个基于 Windows Presentation Foundation（WPF）的应用程序，你想捕获和记录其中

任何线程上任何未经处理的异常。

5.8.2 解决方案

为了挂接必要的事件处理程序，以捕获 WPF 应用程序中所有未处理的异常，可以在应用程序的 App.xaml 文件中添加以下代码。

```
<Application x:Class="UnhandledWPFException.App"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   StartupUri="Window1.xaml"
   DispatcherUnhandledException="Application_DispatcherUnhandledException">
   <Application.MainWindow>
       <Window />
   </Application.MainWindow>
   <Application.Resources>
   </Application.Resources>
</Application>
```

然后，在对应的 App.xaml.cs 代码文件中，添加 `Application_DispatcherUnhandledException` 方法，用于处理未处经理的异常。

```
private void Application_DispatcherUnhandledException(object sender,
          System.Windows.Threading.DispatcherUnhandledExceptionEventArgs e)
{
    // 将异常信息记录到事件日志中
    EventLog.WriteEntry("UnhandledWPFException Application",
        e.Exception.ToString(), EventLogEntryType.Error);
    // 通知用户发生的异常
    MessageBox.Show("Application_DispatcherUnhandledException: " +
        e.Exception.ToString());
    // 标明已经处理了异常
    e.Handled = true;
    // 关闭应用程序
    this.Shutdown();
}
```

5.8.3 讨论

Windows Presentation Foundation 为 .NET 平台创建基于 Windows 的应用程序提供了另外一种方法。为了防止用户看到不雅观的未经处理的异常，需要在 WPF 中编写一些代码，如同在 WinForms 中一样［参见范例 5.7（即 5.7 节），了解如何在 WinForms 中实现这一点］。

`System.Windows.Application` 类是基于 WPF 的应用程序的基类，通过其 `DispatcherUnhandledException` 事件可以处理未经处理的异常。通过在 App.xaml 文件中指定处理事件的方法来建立这个事件处理程序，如下所示。

```
DispatcherUnhandledException="Application_DispatcherUnhandledException">
```

也可以直接在代码中建立这个事件处理程序，而不是采用 XAML 方式，通过在 XAML 文件中添加 `Startup` 事件处理程序（Microsoft 建议在 WPF 应用程序中的此处添加初始化代

码），如下所示。

```
<Application x:Class="UnhandledWPFException.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Window1.xaml"
  Startup="Application_Startup" >
    <Application.MainWindow>
        <Window />
    </Application.MainWindow>
    <Application.Resources>

    </Application.Resources>
</Application>
```

在 `Startup` 事件中，为 `DispatcherUnhandledException` 建立事件处理程序，如下所示。

```
private void Application_Startup(object sender, StartupEventArgs e)
{
    this.DispatcherUnhandledException +=
          new System.Windows.Threading.DispatcherUnhandledExceptionEventHandler(
               Application_DispatcherUnhandledException);
}
```

这非常适合为 WPF 应用程序处理异常，只需挂接事件并获取传送给单个处理程序的所有未处理的异常即可，对吧？不对。就像 WinForms 应用程序中所需要的那样，如果有任何代码在任何非 UI 线程上运行（几乎总会出现这种情况），仍然需要为 `AppDomain` 的 `AppDomain.UnhandledException` 事件挂接处理程序，用于捕获非 UI 线程上的那些异常。为了实现此操作，对 App.xaml.cs 文件进行以下更新。

```
/// <summary>
/// Interaction logic for App.xaml
/// </summary>
public partial class App : Application
{
    private void Application_DispatcherUnhandledException(object sender,
          System.Windows.Threading.DispatcherUnhandledExceptionEventArgs e)
    {
        // 标明已经处理了异常
        e.Handled = true;
        ReportUnhandledException(e.Exception);
    }

    private void Application_Startup(object sender, StartupEventArgs e)
    {
        // WPF UI异常
        this.DispatcherUnhandledException +=
          new System.Windows.Threading.DispatcherUnhandledExceptionEventHandler(
            Application_DispatcherUnhandledException);

        // 那些线程上的异常
        AppDomain.CurrentDomain.UnhandledException +=
            new UnhandledExceptionEventHandler(CurrentDomain_UnhandledException);
    }
```

```
    private void CurrentDomain_UnhandledException(object sender,
                                UnhandledExceptionEventArgs e)
    {
        ReportUnhandledException(e.ExceptionObject as Exception);
    }

    private void ReportUnhandledException(Exception ex)
    {
        // 将异常信息记录到事件日志中
        EventLog.WriteEntry("UnhandledWPFException Application",
            ex.ToString(), EventLogEntryType.Error);
        // 通知用户发生的异常
        MessageBox.Show("Unhandled Exception: " + ex.ToString());
        // 关闭应用程序
        this.Shutdown();
    }
}
```

5.8.4 参考

范例 5.7（即 5.7 节），以及 MSDN 文档中的“DispatcherUnhandledException 事件”和“AppDomain.UnhandledException 事件”主题。

5.9 确定一个进程是否停止了响应

5.9.1 问题

你需要监视一个或多个进程，以确定用户界面是否停止了对系统的响应。这一功能类似于 Task Manager 中基于应用程序的状态显示文本 Responding 或 Not Responding 的那一列。

5.9.2 解决方案

使用例 5-5 中所示的 GetProcessState 方法和 ProcessRespondingState 枚举，确定进程是否停止了响应。

例 5-5：确定进程是否停止了响应

```
public enum ProcessRespondingState
{
    Responding,
    NotResponding,
    Unknown
}

public static ProcessRespondingState GetProcessState(Process p)
{
    if (p.MainWindowHandle == IntPtr.Zero)
    {
        Trace.WriteLine($"{p.ProcessName} does not have a MainWindowHandle");
        return ProcessRespondingState.Unknown;
```

```
        }
        else
        {
            // 这一进程拥有一个MainWindowHandle
            if (!p.Responding)
                return ProcessRespondingState.NotResponding;
            else
                return ProcessRespondingState.Responding;
        }
    }
```

5.9.3 讨论

GetProcessState 方法接受标识一个进程的单一参数 process。然后在 process 参数表示的 Process 对象上调用 Responding 属性。该方法返回一个 ProcessRespondingState 枚举值，指示一个进程目前是响应（Responding）、未响应（NotResponding）或者因为没有主窗口句柄而不能确定该进程是否在响应（Unknown）。

如果正在处理的进程没有 MainWindowHandle，则 Responding 属性总是返回 true。像 Idle、spoolsv、Rundll32 和 svchost 这样的进程并没有主窗口句柄，因此 Responding 属性总会为它们返回 true。为了清除这些进程，可以使用 Process 类的 MainWindowHandle 属性，它将为进程返回主窗口的句柄。如果该属性返回 0，进程就没有主窗口。

为了确定一台机器上的所有进程是否都在响应，可以如下所示调用 GetProcessState 方法。

```
var processes = Process.GetProcesses().ToArray();
Array.ForEach(processes, p =>
    {
        var processState = GetProcessState(p);

        switch (processState)
        {
            case ProcessRespondingState.NotResponding:
                Console.WriteLine($"{p.ProcessName} is not responding.");
                break;
            case ProcessRespondingState.Responding:
                Console.WriteLine($"{p.ProcessName} is responding.");
                break;
            case ProcessRespondingState.Unknown:
                Console.WriteLine(
                    $"{p.ProcessName}'s state could not be determined.");
                break;
        }
    });
```

这段代码将遍历系统中目前在运行的所有进程。Process 类的静态方法 GetProcesses 不带参数，并且返回 Process 对象的数组，包含系统中运行的所有进程的信息。然后把每个 Process 对象传入 GetProcessState 方法中，以确定它是否在响应。Process 类上的其他用于获取 Process 对象的的静态方法是 GetProcessById、GetCurrentProcess 和 GetProcessesByName。

5.9.4 参考

MSDN 文档中的“Process 类”主题。

5.10 在应用程序中使用事件日志

5.10.1 问题

你需要给应用程序添加一种能力，用来记录在应用程序中发生的事件，比如启动、关闭、关键错误，甚至违反安全的情况。除了读、写日志之外，你还需要具有在事件日志中创建、清除、关闭和删除日志的能力。

应用程序也许会需要记录多个日志。例如，当应用程序中发生特定的事件（比如启动和关闭）时，应用程序可能使用自定义的日志来跟踪它们。除了自定义的日志之外，应用程序还可能利用事件日志系统中已经构建的安全日志来读、写应用程序中发生的安全事件。

当需要在本地计算机上创建和维护一个日志并且需要在远程机器上创建和维护另一个复制的日志时，对多个日志的支持就会带来方便。这台远程机器可能包含应用程序在每个用户的机器上的所有运行实例的日志。如果在应用程序中违反了安全，管理员可以使用这些日志快速查找发生或发现的任何问题。事实上，应用程序可以在远程管理机器的后台运行，监视从任何用户的机器写到这个日志的特定日志条目。范例 13.6（即 13.6 节）使用一种事件机制来监视写到事件日志的条目，并且可以轻松地用于增强本范例。

5.10.2 解决方案

使用 Microsoft Windows 操作系统中内建的事件日志，来记录非频繁发生的特定事件。

> 不要用大量不同条目填满事件日志，你可以通过启用或禁用追踪对条目进行处理。事件日志必须包含错误信息，其次是非常重要的项目，但并非所有内容都应该写入到事件日志中。明智地选择何时写入事件日志，才能在寻找线索时不需要对所有日志进行排序。

例 5-6 中所示的 AppEvents 类包含在应用程序中创建和使用事件日志所需的所有方法。

例 5-6：创建和使用事件日志

```
using System;
using System.Diagnostics;

public class AppEvents
{
    // 如果在试图读取注册表项(Security log)时遇到SecurityException，
    // 按照如下指引步骤：
    // 1) 打开注册表编辑器(搜索regedit或者在Run提示符下键入regedit并按回车)
    // 2) 查找到以下键：
    //    HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog\Security
    // 3) 右键单击这一项,然后选择Permissions
```

```
// 4) 添加当前登入的用户,并赋予用户Read权限

// 如果在试图写入事件日志时遇到SecurityException
// "Requested registry access is not allowed.",那么说明事件源还没有创建
// 尝试针对自定义事件重新运行EventLogInstaller,
// 对于本示例代码,运行 "%WINDOWS%\Microsoft.NET\Framework\v4.0.30319\
// InstallUtil.exe AppEventsEventLogInstallerApp.dll"
// 如果你刚刚运行过它,可能需要等待一小会儿,直到Windows完成创建
// 并且识别了添加的日志

const string localMachine = ".";
// 构造函数
public AppEvents(string logName) :
    this(logName, Process.GetCurrentProcess().ProcessName)
{ }

public AppEvents(string logName, string source) :
    this(logName, source, localMachine)
{ }

public AppEvents(string logName, string source,
    string machineName = localMachine)
{
    this.LogName = logName;
    this.SourceName = source;
    this.MachineName = machineName;

    Log = new EventLog(LogName, MachineName, SourceName);
}

private EventLog Log { get; set; } = null;

public string LogName { get; set; }

public string SourceName { get; set; }

public string MachineName { get; set; } = localMachine;

// 方法
public void WriteToLog(string message, EventLogEntryType type,
    CategoryType category, EventIDType eventID)
{
    if (Log == null)
        throw (new ArgumentNullException(nameof(Log),
            "This Event Log has not been opened or has been closed."));

    EventLogPermission evtPermission =
        new EventLogPermission(EventLogPermissionAccess.Write, MachineName);
    evtPermission.Demand();

    // 如果此处遇到SecurityException,参与类顶部的注解
    Log.WriteEntry(message, type, (int)eventID, (short)category);
}

public void WriteToLog(string message, EventLogEntryType type,
```

```
    CategoryType category, EventIDType eventID, byte[] rawData)
{
    if (Log == null)
        throw (new ArgumentNullException(nameof(Log),
            "This Event Log has not been opened or has been closed."));

    EventLogPermission evtPermission =
        new EventLogPermission(EventLogPermissionAccess.Write, MachineName);
    evtPermission.Demand();

    // 如果此处遇到SecurityException,参见类顶部的注解
    Log.WriteEntry(message, type, (int)eventID, (short)category, rawData);
}

public IEnumerable<EventLogEntry> GetEntries()
{
    EventLogPermission evtPermission =
        new EventLogPermission(EventLogPermissionAccess.Administer, MachineName);
    evtPermission.Demand();
    return Log?.Entries.Cast<EventLogEntry>().Where(evt =>
        evt.Source == SourceName);
}

public void ClearLog()
{
    EventLogPermission evtPermission =
        new EventLogPermission(EventLogPermissionAccess.Administer, MachineName);
    evtPermission.Demand();
    if (!IsNonCustomLog())
        Log?.Clear();
}

public void CloseLog()
{
    Log?.Close();
    Log = null;
}

public void DeleteLog()
{
    if (!IsNonCustomLog())
        if (EventLog.Exists(LogName, MachineName))
            EventLog.Delete(LogName, MachineName);
    CloseLog();
}

public bool IsNonCustomLog()
{
    // 因为Application、Setup、Security、System和其他非用户定义的日志
    // 包含关键信息,所以不能删除或清空它们
    if (LogName == string.Empty || // 与Application相同
        LogName == "Application" ||
        LogName == "Security" ||
        LogName == "Setup" ||
        LogName == "System")
```

```
            {
                return true;
            }
            return false;
        }
    }
```

该类中使用的 EventIDType 和 CategoryType 枚举定义如下所示。

```
    public enum EventIDType
    {
        NA = 0,
        Read = 1,
        Write = 2,
        ExceptionThrown = 3,
        BufferOverflowCondition = 4,
        SecurityFailure = 5,
        SecurityPotentiallyCompromised = 6
    }

    public enum CategoryType : short
    {
        None = 0,
        WriteToDB = 1,
        ReadFromDB = 2,
        WriteToFile = 3,
        ReadFromFile = 4,
        AppStartUp = 5,
        AppShutDown = 6,
        UserInput = 7
    }
```

最后要说明的一点是，EventIDType 和 CategoryType 枚举主要设计用于记录违反安全的事件以及对应用程序安全的潜在攻击。使用这些事件 ID 和类别，管理员可以更轻松地跟踪潜在的安全威胁，并在发生违反安全的事件后进行事后分析。你可以轻松地修改这些枚举或者用自己的枚举替换它们，以便跟踪在应用程序运行时发生的不同事件。

5.10.3 讨论

为本范例创建的 AppEvents 类给应用程序提供了易于使用的接口，用于在应用程序中创建、使用和删除一个或多个事件日志。下面描述了 AppEvents 类的方法。

- WriteToLog

  重载该方法允许使用或不用一个包含原始数据的字节数组将条目写到事件日志。

- GetEntries

  返回一个包含事件源中所有事件日志条目的 IEnumerable<EventLogEntry>。

- ClearLog

  如果是一个自定义日志，从该事件日志中删除所有事件日志条目。

- CloseLog

  关闭该事件日志，防止与它的进一步交互。

- DeleteLog

  如果是一个自定义日志，删除该事件日志。

可以把一个 AppEvents 对象添加到包含其他 AppEvents 对象的数组或集合中，每个 AppEvents 对象都对应一个特定的事件日志。下面的代码创建了两个 AppEvents 类，并将它们添加到 ListDictionary 集合中。

```
public void CreateMultipleLogs()
{
    AppEvents AppEventLog = new AppEvents("AppLog", "AppLocal");
    AppEvents GlobalEventLog = new AppEvents("AppSystemLog", "AppGlobal");

    ListDictionary LogList = new ListDictionary();
    LogList.Add(AppEventLog.Name, AppEventLog);
    LogList.Add(GlobalEventLog.Name, GlobalEventLog);
```

为了写到其中任何一个日志中，可以按名称从 ListDictionary 对象中获取 AppEvents 对象，把得到的对象类型转换成 AppEvents 类型，并且调用 WriteToLog 方法。

```
((AppEvents)LogList[AppEventLog.Name]).WriteToLog("App startup",
    EventLogEntryType.Information, CategoryType.AppStartUp,
    EventIDType.ExceptionThrown);

((AppEvents)LogList[GlobalEventLog.Name]).WriteToLog(
   "App startup security check",
    EventLogEntryType.Information, CategoryType.AppStartUp,
    EventIDType.BufferOverflowCondition);
```

通过在 ListDictionary 对象中包含所有 AppEvents 对象，可以轻松地遍历应用程序已经实例化的所有 AppEvents。使用 foreach 循环，可以把一条消息同时写到本地和远程事件日志中。

```
foreach (DictionaryEntry Log in LogList)
{
    ((AppEvents)Log.Value).WriteToLog("App startup",
        EventLogEntryType.FailureAudit,
        CategoryType.AppStartUp, EventIDType.SecurityFailure);
}
```

要删除 LogList 对象中的每个日志，可以使用下面的 foreach 循环。

```
foreach (DictionaryEntry Log in LogList)
{
    ((AppEvents)Log.Value).DeleteLog();
}
    LogList.Clear();
```

你需要知道几个关键的要点。第一个涉及关于构造多个 AppEvents 类的一个小问题。如果创建两个 AppEvents 对象，并且把相同的 source 字符串传入 AppEvents 构造函数中，将会引发一个异常。考虑下面的代码，它利用相同的 source 字符串实例化两个 AppEvents 对象。

```
AppEvents appEventLog = new AppEvents("AppLog", "AppLocal");
AppEvents globalEventLog = new AppEvents("Application", "AppLocal");
```

在实例化对象时没有发生错误，但是当对 globalEventLog 对象调用 WriteToLog 方法时，将会引发下面的异常。

```
An unhandled exception of type 'System.ArgumentException' occurred in system.dll.

Additional information: The source 'AppLocal' is not registered in log
'Application'. (It is registered in log 'AppLog'.) " The Source and Log
properties must be matched, or you may set Log to the empty string, and
it will automatically be matched to the Source property.
```

之所以会发生这个异常，是因为 WriteToLog 方法在内部调用 EventLog 对象的 WriteEntry 方法。WriteEntry 方法在内部检查指定的源是否被注册到你尝试写的日志。在这种情况下，AppLog 源被注册到分配给它的第一个日志，即 AppLog 日志。如果第二次尝试把这个相同的源注册到另一个日志，即 Application 日志，则会悄然失败。在尝试使用 EventLog 对象的 WriteEntry 方法之前，不会知道这次尝试已经失败。

关于 AppEvents 类的另一个关键点是如下代码，它位于每个方法（DeleteLog 方法除外）的开始处。

```
if (log == null)
    throw (new ArgumentNullException("log",
        "This Event Log has not been opened or has been closed."))
```

这段代码检查私有成员变量 log 是否是一个 null 引用。如果是，就会引发一个 ArgumentException，通知该类的用户在创建 EventLog 对象时发生了一个问题。DeleteLog 方法不会检查变量 log 是否为 null，因为它会删除事件日志源和事件日志本身。除了在该方法的末尾外，在这个过程中不会涉及 EventLog 对象；其中如果 log 还不是 null 的话，它会被关闭并被设置为 null。不管变量 log 的状态是什么，都应该在此方法中删除源和事件日志。

ClearLog 和 DeleteLog 方法在确定是否删除日志时会作出一个关键的选择。下面的代码会阻止从系统中删除应用程序、安全和系统事件日志。

```
public bool IsNonCustomLog()
{
    // 因为Application、Setup、Security、System和其他非用户定义的日志
    // 包含关键信息,所以不能删除或清空它们
    if (LogName == string.Empty || // 与Application相同
        LogName == "Application" ||
        LogName == "Security" ||
        LogName == "Setup" ||
        LogName == "System")
    {
        return true;
    }
    return false;
}
```

如果删除了其中的任何日志，也会删除使用该特定日志注册的源。一旦删除了日志，就会

永久删除；相信我们，在没有备份的情况下尝试重建日志及其源并不是一件有趣的事情。

尽管如此，为了使 `AppEvents` 类工作，首先需要创建事件源。事件日志使用事件源来确定是哪个应用程序记录的事件。仅在管理上下文中运行时，才可以建立事件源。有下面两种方法可以做到这一点。

- 调用 `EventLog.CreateEventSource` 方法。
- 使用一个 `EventLogInstaller`。

虽然可以创建一个控制台应用程序，调用 `CreateEventSource` 方法，并让一个用户在其机器的管理上下文中运行，但推荐的选项是建立一个 `EventLogInstaller` 类，它可以用于 InstallUtil.exe（由 .NET Framework 提供），以创建初始的事件源和自定义日志。

下面展示的 `AppEventsEventLogInstaller` 将为我们建立事件日志和源。此安装程序不仅可以由 InstallUtil 调用，还可以由大多数安装包调用。因此，可以将其加入你喜爱的安装软件，只要用户具有管理访问权限（或至少有拥有相应权限的 IT 专业人员帮助），就能在安装时注册事件日志和源。

```
/// <summary>
/// 安装：C:\Windows\Microsoft.NET\Framework\v4.0.30319\InstallUtil.exe
/// [PathToBinary]\AppEventsEventLogInstallerApp.dll
/// 卸载：C:\Windows\Microsoft.NET\Framework\v4.0.30319\InstallUtil.exe -u
/// [PathToBinary]\AppEventsEventLogInstallerApp.dll
/// </summary>
[RunInstaller(true)]
public class AppEventsEventLogInstaller : Installer
{
    private EventLogInstaller evtLogInstaller;

    public AppEventsEventLogInstaller()
    {
        evtLogInstaller = new EventLogInstaller();
        evtLogInstaller.Source = "APPEVENTSSOURCE";
        evtLogInstaller.Log = ""; // 默认为Application
        Installers.Add(evtLogInstaller);

        evtLogInstaller = new EventLogInstaller();
        evtLogInstaller.Source = "AppLocal";
        evtLogInstaller.Log = "AppLog";
        Installers.Add(evtLogInstaller);

        evtLogInstaller = new EventLogInstaller();
        evtLogInstaller.Source = "AppGlobal";
        evtLogInstaller.Log = "AppSystemLog";
        Installers.Add(evtLogInstaller);
    }
    public static void Main()
    {
        AppEventsEventLogInstaller appEventsEventLogInstaller =
            new AppEventsEventLogInstaller();
    }
}
```

如果你使用 `InstallUtil` 在本机如此设置，当使用适当的管理上下文（“Run As Administrator”）时，可能会看到如下输出。

```
C:\Windows\Microsoft.NET\Framework\v4.0.30319\InstallUtil.exe
C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
AppEventsEventLogInstallerApp.dll


C:\WINDOWS\system32>C:\Windows\Microsoft.NET\Framework\v4.0.30319\InstallUtil.exe
C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
AppEventsEventLogInstallerApp.dll
Microsoft (R) .NET Framework Installation utility Version 4.0.30319.33440
Copyright (C) Microsoft Corporation.  All rights reserved.


Running a transacted installation.

Beginning the Install phase of the installation.
See the contents of the log file for the
C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
AppEventsEventLogInstallerApp.dll
assembly's progress.
The file is located at C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\

AppEventsEventLogInstallerApp.InstallLog.
Installing assembly 'C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
AppEventsEventLogInstallerApp.dll'.
Affected parameters are:
   logtoconsole =
   logfile = C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
AppEventsEventLogInstallerApp.InstallLog
   assemblypath =
C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
AppEventsEventLogInstallerApp.dll
Creating EventLog source APPEVENTSSOURCE in log ...
Creating EventLog source AppLocal in log AppLog...
Creating EventLog source AppGlobal in log AppSystemLog...

The Install phase completed successfully, and the Commit phase is beginning.
See the contents of the log file for the
C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
AppEventsEventLogInstallerApp.dll
assembly's progress.
The file is located at C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
AppEventsEventLogInstallerApp.InstallLog.
Committing assembly
'C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
AppEventsEventLogInstallerApp.dll'.
Affected parameters are:
   logtoconsole =
   logfile = C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
AppEventsEventLogInstallerApp.InstallLog
   assemblypath =
C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
```

```
AppEventsEventLogInstallerApp.dll

The Commit phase completed successfully.

The transacted install has completed.
```

如果试图在非管理上下文中运行 InstallUtil，将会看到如下的结果。

```
C:\Windows\Microsoft.NET\Framework\v4.0.30319\InstallUtil.exe
C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
AppEventsEventLogInstallerApp.dll
Microsoft (R) .NET Framework Installation utility Version 4.0.30319.33440
Copyright (C) Microsoft Corporation.  All rights reserved.


Running a transacted installation.

Beginning the Install phase of the installation.
See the contents of the log file for the
C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\AppEventsEventLogInstallerApp.dll
assembly's progress.
The file is located at C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\

AppEventsEventLogInstallerApp.InstallLog.
Installing assembly
'C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
AppEventsEventLogInstallerApp.dll'.
Affected parameters are:
   logtoconsole =
   logfile = C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
AppEventsEventLogInstallerApp.InstallLog
   assemblypath =
C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
AppEventsEventLogInstallerApp.dll
Creating EventLog source APPEVENTSSOURCE in log APPEVENTSLOG...

An exception occurred during the Install phase.
System.Security.SecurityException: The source was not found, but some or all
event logs could not be searched. Inaccessible logs: Security.

The Rollback phase of the installation is beginning.
See the contents of the log file for the
C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
AppEventsEventLogInstallerApp.dll
assembly's progress.
The file is located at C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
AppEventsEventLogInstallerApp.InstallLog.
Rolling back assembly
'C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
AppEventsEventLogInstallerApp.dll'.
Affected parameters are:
   logtoconsole =
   logfile = C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
AppEventsEventLogInstallerApp.InstallLog
   assemblypath =
```

```
C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
AppEventsEventLogInstallerApp.dll
Restoring event log to previous state for source APPEVENTSSOURCE.
An exception occurred during the Rollback phase of the
System.Diagnostics.EventLogInstaller installer.
System.Security.SecurityException: Requested registry access is not allowed.
An exception occurred during the Rollback phase of the installation. This except
ion will be ignored and the rollback will continue. However, the machine might n
ot fully revert to its initial state after the rollback is complete.

The Rollback phase completed successfully.

The transacted install has completed.
The installation failed, and the rollback has been performed.
```

InstallUtil 不仅可以安装事件日志和源，还可以使用 -u 参数帮助移除它们。

```
C:\Windows\Microsoft.NET\Framework\v4.0.30319\InstallUtil.exe -u
C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
AppEventsEventLogInstallerApp.dll

Microsoft (R) .NET Framework Installation utility Version 4.0.30319.33440
Copyright (C) Microsoft Corporation.  All rights reserved.

The uninstall is beginning.
See the contents of the log file for the
C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
AppEventsEventLogInstallerApp.dll
assembly's progress.
The file is located at C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
AppEventsEventLogInstallerApp.InstallLog.
Uninstalling assembly
'C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
AppEventsEventLogInstallerApp.dll'.
Affected parameters are:
   logtoconsole =
   logfile = C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
AppEventsEventLogInstallerApp.InstallLog
   assemblypath =
C:\CSCB6\AppEventsEventLogInstallerApp\bin\Debug\
AppEventsEventLogInstallerApp.dll
Removing EventLog source AppGlobal.
Deleting event log AppSystemLog.
Removing EventLog source AppLocal.
Deleting event log AppLog.
Removing EventLog source APPEVENTSSOURCE.

The uninstall has completed.
```

应该把从应用程序写到事件日志中的条目数量减至最少，因为写到事件日志中会导致性能损失，并且没有将某些日志设置为达到一定数量的条目后滚动或清除。向事件日志写入太多信息可能会显著减慢应用程序或者导致服务器问题。明智地选择写到事件日志的条目。

5.10.4 参考

MSDN 文档中的“EventLog 类”“InstallUtil.exe”和“EventLogInstaller 类”主题。

5.11 监视事件日志中的特定条目

5.11.1 问题

你可能有多个应用程序写到一个事件日志。对于其中每个应用程序，你想使用一个监控应用程序来监视写到事件日志中的一个或多个特定的日志条目。例如，你可能想监视一个日志条目，它指示应用程序遇到关键错误或者意外关闭。应该实时报告这些日志条目。

5.11.2 解决方案

监视事件日志中的特定条目需要执行以下步骤。

(1) 创建以下方法来建立事件处理程序，以处理事件日志写操作：

```
public void WatchForAppEvent(EventLog log)
{
    log.EnableRaisingEvents = true;
    // 挂接System.Diagnostics.EntryWrittenEventHandler
    log.EntryWritten += new EntryWrittenEventHandler(OnEntryWritten);
}
```

(2) 创建事件处理程序检查日志条目，并确定是否执行进一步的动作。例如：

```
public static void OnEntryWritten(object source,
                                  EntryWrittenEventArgs entryArg)
{
    if (entryArg.Entry.EntryType == EventLogEntryType.Error)
    {
        Console.WriteLine(entryArg.Entry.Message);
        Console.WriteLine(entryArg.Entry.Category);
        Console.WriteLine(entryArg.Entry.EntryType.ToString());
        // 需要时执行进一步的动作
    }
}
```

5.11.3 讨论

本范例涉及 EntryWrittenEventHandler 委托，无论何时把一个新条目写到事件日志中，它都会回调一个方法。EntryWrittenEventHandler 委托接受两个参数：object 类型的 source 和 EntryWrittenEventArgs 类型的 entryArg。entryArg 参数是这两个参数中更有趣的参数。它包含一个名为 Entry 的属性，该属性返回一个 EventLogEntry 对象。这个 EventLogEntry 对象包含所有你需要的关于写到事件日志的条目的信息。

将你正在监视的这个事件日志作为 WatchForAppEvent 方法的参数进行传递。该方法执行两个操作。首先，它将 log 的 EnableRaisingEvents 属性设置为 true。如果把该属性设置为

false，那么在把条目写到这个事件日志中时将不会为它引发任何事件。该方法执行的第二个操作是把 OnEntryWritten 回调方法添加到这个事件日志的事件处理程序的列表中。

为了阻止该委托调用 OnEntryWritten 回调方法，可以把 EnableRaisingEvents 属性设置为 false，从而有效地关闭委托。

注意传递给 OnEntryWritten 回调方法的 entryArg 参数的 Entry 对象是只读的，因此在把条目写到事件日志中之前不能修改它。

5.11.4 参考

MSDN 文档中的“EventLog.EntryWritten 事件”主题。

5.12 实现一个简单的性能计数器

5.12.1 问题

你需要使用一个性能计数器来跟踪特定于应用程序的信息。例如，较为简单的性能计数器可以找出连续取样之间的计数器值中的变化，或者只是统计某种动作发生的次数。还有其他更复杂的计数器，但是本范例中没有涉及它们。例如，可以构建一个自定义计数器，用于记录数据库事务的数量、连接到服务器的网络连接失败的次数，甚至记录每分钟连接到你的 Web 服务的用户数。

5.12.2 解决方案

创建一个简单的性能计数器，例如找出连续取样之间的计数器值中的变化或者只是统计某种动作发生的次数。使用下面的方法（CreateSimpleCounter）创建一个简单的自定义计数器。

```
public static PerformanceCounter CreateSimpleCounter(string counterName,
    string counterHelp, PerformanceCounterType counterType, string categoryName,
    string categoryHelp)
{
    CounterCreationDataCollection counterCollection =
        new CounterCreationDataCollection();

    // 创建一个自定义计数器对象并将其添加到计数器集合中
    CounterCreationData counter =
        new CounterCreationData(counterName, counterHelp, counterType);
    counterCollection.Add(counter);

    // 创建类别
    if (PerformanceCounterCategory.Exists(categoryName))
        PerformanceCounterCategory.Delete(categoryName);

    PerformanceCounterCategory appCategory =
        PerformanceCounterCategory.Create(categoryName, categoryHelp,
            PerformanceCounterCategoryType.SingleInstance, counterCollection);
```

```
        // 创建计数器并将其初始化
        PerformanceCounter appCounter =
            new PerformanceCounter(categoryName, counterName, false);

        appCounter.RawValue = 0;

        return (appCounter);
    }
```

5.12.3 讨论

该方法执行的第一个操作是创建 CounterCreationDataCollection 对象和 CounterCreationData 对象。使用传递给 CreateSimpleCounter 方法的 counterName、counterHelp 和 countertype 参数创建 CounterCreationData 对象。然后把该 CounterCreationData 对象添加到 counterCollection 中。

> 默认情况下，ASPNET 用户账号以及许多其他用户账号会因安全原因阻止读取性能计数器。你可以提升这些账户的权限，也可以模拟具有访问权限的账户来启用该功能。不过，这在以后将变成应用程序的一项部署要求。
>
> 这样做也存在风险，因为作为开发人员，你是安全事项的第一道防线。如果你建立应用，并作出放松安全限制的选择，那么将承担这一责任，所以请不要不加选择地或者在没有充分理解后果的情况下就这样做。

如果系统上没有注册 categoryName（它是一个包含类别名称的字符串，作为参数传到个方法），就会用 PerfomanceCounterCategory 对象创建一个新类别。如果类别被注册过了，就会删除它并重新创建一个。最后，从 PerfomanceCounter 对象创建实际的性能计数器。该对象被初始化为 0 并由方法返回。PerfomanceCounterCategory 获取一个 PerfomanceCounterCategoryType 作为参数。可能的设置如表 5-1 所示。

表5-1：PerformanceCounterCategoryType枚举值

名　称	描　述
MultiInstance	性能计数器可以有多个实例
SingleInstance	性能计数器只能有一个实例
Unknown	该性能计数器的实例功能是未知的

CreateSimpleCounter 方法返回一个将由应用程序使用的 PerformanceCounter 对象。应用程序可以在 PerformanceCounter 对象上执行多种操作。应用程序可以使用以下三种方法之一递增或递减它。

```
    long value = appCounter.Increment();
    long value = appCounter.Decrement();
    long value = appCounter.IncrementBy(i);
    // 此外,可以把一个负数传递到
    // IncrementBy方法以模拟DecrementBy方法
    //（此方法并未包含在此类中）
```

```
// 例如：
long value = appCounter.IncrementBy(-i);
```

前两个方法不接受任何参数，而第三个方法接受一个 long 类型的参数，其中包含用于递增计数器的数字。这三个方法都返回一个 long 类型，指示计数器的新值。

除了递增或递减计数器之外，还可以获取计数器在应用程序中多个时间点的样本。样本是计数器及其所有值在特定时刻的快照。可以使用下面一行代码获取样本。

```
CounterSample counterSampleValue = appCounter.NextSample();
```

NextSample 方法不接受任何参数，并且返回一个 CounterSample 结构。

在应用程序中的另一个时间点，可以再次对计数器取样，并且可以把两个样本传递给 CounterSample 类的静态方法 Calculate。可以利用单独一行代码执行此操作，如下所示。

```
float calculatedSample = CounterSample.Calculate(counterSampleValue,
                                                 appCounter.NextSample());
```

可以对计算得到的样本 calculatedSample 进行存储，以便将来分析。

.NET Framework 中已经提供的简单性能计数器包括以下这些。

- CounterDelta32/CounterDelta64

  确定计数器的两次取样之间值的区别（或变化）。CounterDelta64 计数器可以保存比 CounterDelta32 更大的值。

- CounterTimer

  计算 CounterTimer 值随着 CounterTimer 时间的改变而发生的改变的百分比。以总采样时间的百分比跟踪资源的平均活跃时间。

- CounterTimerInverse

  计算 CounterTimer 计数器的倒数。以总采样时间的百分比跟踪资源的平均非活跃时间。

- CountPerTimeInterval32/CountPerTimeInterval64

  计算一段时间内队列中等待资源的项目数。这些计数器会给出按间隔持续时间划分的最后两个样本间隔内的队列长度增量。

- ElapsedTime

  计算计数器记录的事件开始时间与当前时间之差，以秒为单位。

- NumberOfItems32/NumberOfItems64

  这些计数器以十进制格式返回它们的值。NumberOfItems64 计数器可以保存比 NumberOfItems32 更大的值。该计数器不需要传递给 CounterSample 类的静态方法 Calculate；没有需要计算的值。使用 PerformanceCounter 对象的 RawValue 属性来代替（也就是说，在本范例中将使用 appCounter.RawValue 属性）。

- RateOfCountsPerSecond32/RateOfCountsPerSecond64

  计算 RateOfCountsPerSecond* 值随着 RateOfCountsPerSecond* 时间的改变而发生的改变，以秒为单位。RateOfCountsPerSecond64 计数器可以保存比 RateOfCountsPerSecond32 计数

器更大的值。

- `Timer100Ns`

  将活动组件的时间显示为样本间隔的总消耗时间的百分比，以 100 纳秒（ns）为单位。这类计数器的一个例子是“Processor\ % User Time”。

- `Timer100nsInverse`

  一个基于百分比的计数器，显示在样本时间间隔期间跟踪的平均活动时间的百分比。这类计数器的一个例子是“Processor\ % Processor Time”。

5.12.4 参考

MSDN 文档中的“`PerformanceCounter` 类”“`PerformanceCounterType` 枚举”“`PerformanceCounterCategory` 类”“ASP.NET Impersonation”和“Monitoring Performance Thresholds”主题。

5.13 为类创建自定义的调试显示

5.13.1 问题

你在应用程序中使用了一组类。你想在调试器中快速查看类中保存的特定实例。默认的调试器显示不会展示自定义类的任何有用信息。

5.13.2 解决方案

向类添加一个 `DebuggerDisplayAttribute`，使调试器显示你认为有用的此类相关信息。例如，如果有一个保存尊称和姓名的 `Citizen` 类，就可以像下面这样添加一个 `DebuggerDisplayAttribute`。

```
[DebuggerDisplay("Citizen Full Name = {Honorific}{First}{Middle}{Last}")]
public class Citizen
{
    public string Honorific { get; set; }
    public string First { get; set; }
    public string Middle { get; set; }
    public string Last { get; set; }
}
```

现在，当实例化 `Citizen` 类的实例时，调试器将按此类的 `DebuggerDisplayAttribute` 指导它的方式显示信息。为了查看此行为，可实例化两个 `Citizen`（Mrs. Alice G. Jones 和 Mr. Robert Frederick Jones），如下所示。

```
Citizen mrsJones = new Citizen()
{
    Honorific = "Mrs.",
    First = "Alice",
    Middle = "G.",
    Last = "Jones"
```

```
};
Citizen mrJones = new Citizen()
{
    Honorific = "Mr.",
    First = "Robert",
    Middle = "Frederick",
    Last = "Jones"
};
```

在调试器下运行这段代码时，就会使用自定义的显示，如图 5-9 所示。

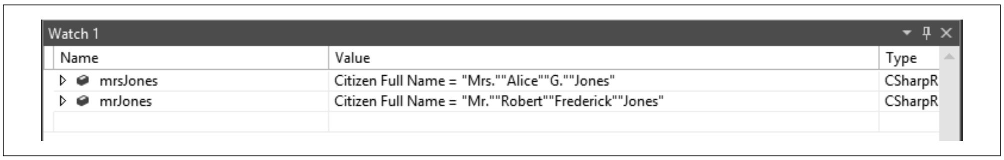


图 5-9：DebuggerDisplayAttribute 控制的调试器显示

5.13.3　讨论

能够迅速查看你编写的类的相关信息当然很好，但是这种特性的更强大的功能在于它能够让你的团队成员迅速了解类实例保存的是什么。可以从 DebuggerDisplayAttribute 声明访问 this 指针，但是使用 this 指针访问的任何属性在处理前都不会对属性的特性进行求值。实质上，如果访问作为构造显示字符串的一部分的当前对象实例的属性，并且该属性具有特性，则不会处理这些特性，因此也许没有获取到预想中的值。如果你已经具有自定义的 ToString() 重写版本，调试器将把它们用作 DebuggerDisplayAttribute；无需指定它们，只要在 Tools\Options\Debugging 下启用了正确的选项即可，如图 5-10 所示。

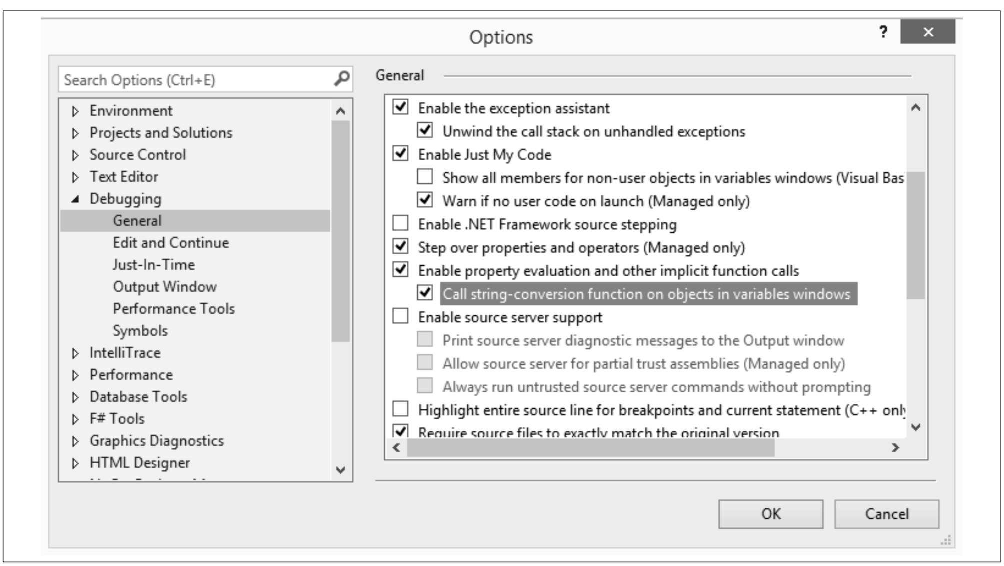


图 5-10：将调试器设置为调用 ToString() 以进行对象显示

5.13.4 参考

MSDN 文档中的“使用 DebuggerDisplay 特性”和“DebuggerDisplayAttribute”主题。

5.14 跟踪异常从何而来

5.14.1 问题

你想确定异常是在什么方法中被捕获的，或者导致异常引发的方法是被谁调用的，以帮助调试问题。

5.14.2 解决方案

使用位于 System.Runtime.CompilerServices 命名空间中的 CallerMemberName、CallerFilePath 和 CallerLineNumber 特性（也被称为调用方信息特性）来确定调用者方法。

例如，如果需要记录捕换一个异常的 catch 语句块的位置，可以使用一个类似 RecordCatchBlock 的方法，代码如下所示。

```
public void RecordCatchBlock(Exception ex,
    [CallerMemberName] string memberName = "",
    [CallerFilePath] string sourceFilePath = "",
    [CallerLineNumber] int sourceLineNumber = 0)
{
    string catchDetails =
        $"{ex.GetType().Name} caught in member \"{memberName}\" " +
        $"in catch block encompassing line {sourceLineNumber} " +
        $"in file {sourceFilePath} " +
        $"with message \"{ex.Message}\"";
    Console.WriteLine(catchDetails);
}
```

然后在 catch 语句块中调用此方法，如下所示。

```
public void TestCallerInfoAttribs()
{
    try
    {
        LibraryMethod();
    }
    catch(Exception ex)
    {
        RecordCatchBlock(ex);
    }
}
```

这使得你可以看到捕获到的异常的类型，异常被捕获时所在的类成员名称，异常被捕获时所在的源码文件和 catch 语句块的行号，而不需要遍历调用堆栈。

```
LibraryException caught in member "TestCallerInfoAttribs" in catch block encompa
ssing line 1303 in file C:\CSCB6\CSharpRecipes\
```

```
05_DebuggingAndExceptionHandling.cs
with message "Object reference not set to an instance of an object."
```

还可以使用这些特性来帮助确定什么方法调用了库方法，因为有时候难以调试出哪个函数调用了库方法。

```
public void LibraryMethod(
            [CallerMemberName] string memberName = "",
            [CallerFilePath] string sourceFilePath = "",
            [CallerLineNumber] int sourceLineNumber = 0)
{
    try
    {
        //执行一些库行为，
        //发生了问题
        throw new NullReferenceException();
    }
    catch(Exception ex)
    {
        //封装异常,捕获库方法被调用的来源
        throw new LibraryException(ex)
        {
            CallerMemberName = memberName,
            CallerFilePath = sourceFilePath,
            CallerLineNumber = sourceLineNumber
        };
    }
}
```

利用 LibraryException，在运行时可以记录调用者方法的特性，与原始异常同时输出。

```
[Serializable]
public class LibraryException : Exception
{
    public LibraryException(Exception inner) : base(inner.Message,inner)
    {
    }
    public string CallerMemberName { get; set; }
    public string CallerFilePath { get; set; }
    public int CallerLineNumber { get; set; }

    public override void GetObjectData(SerializationInfo info,
                                       StreamingContext context)
    {
        base.GetObjectData(info, context);
        info.AddValue("CallerMemberName", this.CallerMemberName);
        info.AddValue("CallerFilePath", this.CallerFilePath);
        info.AddValue("CallerLineNumber", this.CallerLineNumber);
    }

    public override string ToString() => "LibraryException originated in " +
        $"member \"{CallerMemberName}\" " +
        $"on line {CallerLineNumber} " +
        $"in file {CallerFilePath} " +
        $"with exception details: {Environment.NewLine}" +
```

```
            $"{InnerException.ToString()}";
    }
```

LibraryException.ToString 方法提供了问题的摘要。

```
LLibraryException originated in member "TestCallerInfoAttribs" on line 1299 in
file C:\CSCB6\CSharpRecipes\05_DebuggingAndExceptionHandling.cs with exception
details:
System.NullReferenceException: Object reference not set to an instance of an obj
ect.
   at CSharpRecipes.DebuggingAndExceptionHandling.LibraryMethod(String memberNam
e, String sourceFilePath, Int32 sourceLineNumber) in D:\PRJ32\Book_6_0\CS60_Cook
book\CSCB6\CSharpRecipes\05_DebuggingAndExceptionHandling.cs:line 1318
```

5.14.3 讨论

因为 CallerInfo 特性在编译时确定，所以它在运行时不会产生从堆栈中获取前一方法来源信息的成本。虽然不像完整的堆栈跟踪那样详尽，但它是一种更廉价和更简单的替代，可以给你方法、文件和行号信息，你可以借此增强异常日志功能。你在任何时候都能让收到的缺陷 /bug 报告 / 问题单（选择你喜欢的代码出问题时通知到你的方式）中带有这类信息，生活会变得更加轻松。

或许你注意到了 CallerInfo 特性需要一个默认值。

```
public void RecordCatchBlock(Exception ex,
    [CallerMemberName] string memberName = "",
    [CallerFilePath] string sourceFilePath = "",
    [CallerLineNumber] int sourceLineNumber = 0)
```

这些参数需要默认值是因为 CallerInfo 特性是用可选参数实现的，而可选参数需要一个默认值。你仍然能够调用带有特性的方法，不需要提供参数值，如下所示。

```
RecordCatchBlock(ex);
```

5.14.4 参考

MSDN 文档中的“CallerMemberNameAttribute”“CallerFilePathAttribute”和“CallerLineNumberAttribute”主题。

5.15 在异步情境下处理异常

5.15.1 问题

你正在使用 async 和 await 来调用异步方法，需要能够捕获此方法（或多个方法）执行期间引发的任何异常。

5.15.2 解决方案

在处理单个方法调用时出现的异常时，.NET Framework 会处理在异步调用和等待异步返回

之间出现的异常返回；而在处理多个同时调用的异步方法中的异常时，需要一些额外的工作来提取所有的异常细节。最后，在处理一个异常时，可以在 catch 语句块中调用一个异步方法来完成工作。

为了说明这一点，我们来看一个很常见的场景：一个名叫 Bill 的软件开发经理有一些工作需要完成。

用户故事1：Bill需要让Steve实现产品中的一个新功能。

Bill 来到 Steve 的办公桌前，要求他实现这个冲刺（sprint）中的新功能，然后离开，由 Steve 自己来实现（与 Bill 的要求异步进行）。

```
try
{
    // Steve, get that project done!
    await SteveCreateSomeCodeAsync();
}
catch (DefectCreatedException dce)
{
    Console.WriteLine($"Steve introduced a Defect: {dce.Message}");
}
```

Steve 像所有开发人员一样努力工作，但就算是我们中最好的人也会有不顺利的一天。Steve 碰巧在按要求在 SteveCreateSomeCodeAsync 中实现的功能中出现了一个缺陷。幸运的是，尽管 Steve 以异步方式执行此操作，我们仍然能够以通常的方式捕获 DefectCreatedException 并处理它，因为 async 和 await 支持自动将异常传输回到 catch 语句块中。（在 5.15.3 节中有更多关于此种机制如何运行的信息。请查找 ExceptionDispatchInfo！）

来自捕获异常的输出让我们知道问题出在哪里，所以 Steve 晚些时候可以修复它。

```
Steve introduced a Defect: A defect was introduced: (Null Reference on line 42)
```

用户故事2：Bill有大量的功能需要由Jay、Tom和Seth实现

Bill 知道 Steve 很忙，所以他来到团队中其他成员处（Jay、Tom 和 Seth），要求完成这个冲刺中的一些新功能。看起来他们需要周六来加班。Jay、Tom 和 Seth 聚在一起，划分了工作，然后同时开始编程。即使他们可能会在不同的时间完成，Bill 仍然想要查找到他们产生的任何缺陷。

```
// 大伙,这个周末要完成新的功能
// 你们最好快一点
Task jayCode = JayCreateSomeCodeAsync();
Task tomCode = TomCreateSomeCodeAsync();
Task sethCode = SethCreateSomeCodeAsync();

Task teamComplete = Task.WhenAll(new Task[] { jayCode, tomCode, sethCode });
try
{
    await teamComplete;
}
catch
{
```

```
        // 获得动作集合引发异常的信息
        var defectMessages =
            teamComplete.Exception?.InnerExceptions.Select(e =>
                e.Message).ToList();
        defectMessages?.ForEach(m =>
            Console.WriteLine($"{m}"));
    }
```

首先，将每个工作单元（JayCreateSomeCodeAsync、TomCreateSomeCodeAsync 和 SethCreate-SomeCodeAsync）转变为一个 Task。然后调用 Task.WhenAll 方法来创建一个容器 Task（teamComplete），它将在所有单独的 Task 都完成之后完成。

一旦所有任务都完成之后，如果任何一个 Task 在执行过程中出现异常，await 将引发 AggregateException。这个 AggregateException 可以通过 teamComplete.Exception 属性来访问，它包含了一个 InnerExceptions 列表，其类型为 ReadOnlyCollection<Exception>。

因为是开发人员引起了这些异常，所以它们大多是 DefectCreatedExceptions！

生成的日志记录告诉我们团队需要在哪里改正。

```
A defect was introduced: (Ambiguous Match on line 2)
A defect was introduced: (Quota Exceeded on line 11)
A defect was introduced: (Out Of Memory on line 8)
```

用户故事3：Bill想要记录在实现一项新功能时是否存在任何问题

最终，Bill 意识到他需要一个更好的系统来确定是否存在引入到代码中的缺陷。Bill 在代码中添加了日志，当捕获到 DefectCreatedException 时，将缺陷的详细信息写入到 EventLog 中。由于写入 EventLog 可能会影响到性能，他决定以异步方式执行此操作。

```
try
{
    await SteveCreateSomeCodeAsync();
}
catch (DefectCreatedException dce)
{
    await WriteEventLogEntryAsync("ManagerApplication", dce.Message,
        EventLogEntryType.Error);
    throw;
}
```

5.15.3 讨论

以异步方式运行代码并不意味着你不再需要有正确的错误处理，它只是将过程复杂化了一点。幸运的是，Microsoft 的 C# 和 .NET 团队做了大量工作以使得这一任务尽可能轻松一些。

异步等待这些操作意味着它们在一个上下文中运行，要么是当前的 SynchronizationContext，要么是 TaskScheduler。这一上下文在异步方法等待另一个方法时被捕获，之后在异步方法继续工作时恢复。

捕捉的上下文取决于在以下哪个位置执行了异步方法代码。

- 用户界面（WinForms/WPF）：UI 上下文
- ASP.NET：ASP.NET 请求上下文
- 其他：线程池上下文

在用户故事 1 中，我们提到了 async 和 await 的实现用到了一个名为 System.Runtime.ExceptionServices.ExceptionDispatchInfo 的类，用于处理以下情形：在一个线程上引发某个异常，并且作为异步操作的结果，需要在另一个线程上捕获该异常。

使用 ExceptionDispatchInfo 可以捕获在一个线程上引发的异常，然后在另一个线程上重新引发，而不会丢失任何信息（异常数据和堆栈跟踪）。从异常处理的观点上来看，这是 await 一个 async 方法时所发生的事情。

值得注意的另外一点是 ConfigureAwait 的使用，它允许你完成 async 方法之后更改上下文恢复的行为。如果将 false 传递给 ConfigureAwait，它将不会尝试恢复原始上下文。

```
await MyAsyncMethod().ConfigureAwait(false);
```



如果使用 ConfigureAwait(false)，则在 await 完成并且 async 方法恢复之后的任何代码都不能依赖原始的上下文，因为代码继续执行所在的线程没有原始的上下文信息。例如，如果 async 方法在 ASP.NET 的上下文中被调用，那么在它恢复后请求上下文将不再可用。

这个努力工作的团队的代码展示在例 5-7 中。

例 5-7：团队工作

```
public async Task TestHandlingAsyncExceptionsAsync()
{
    // 团队生产软件
    // 经理让Steve去编写代码,引发了"DefectCreatedException"异常
    // 经理让Jay、Tom和Seth去编写代码,都引发了DefectCreatedException

    // 单个asyn方法调用
    try
    {
        // Steve,去完成项目
        await SteveCreateSomeCodeAsync();
    }
    catch (DefectCreatedException dce)
    {
        Console.WriteLine($"Steve introduced a Defect: {dce.Message}");
    }

    // 多个async方法(WaitAll)
    // 大伙,这个周末要完成新的功能
    // 你们最好快一点
    Task jayCode = JayCreateSomeCodeAsync();
    Task tomCode = TomCreateSomeCodeAsync();
    Task sethCode = SethCreateSomeCodeAsync();

    Task teamComplete = Task.WhenAll(new Task[] { jayCode, tomCode, sethCode });
    try
```

```
    {
        await teamComplete;
    }
    catch
    {
        // 获得动作集合引发异常的信息
        var defectMessages =
            teamComplete.Exception?.InnerExceptions.Select(e =>
                e.Message).ToList();
        defectMessages?.ForEach(m =>
            Console.WriteLine($"{m}"));
    }

    // 在异常处理中等待一个动作
    // 讨论原始引发位置如何通过System.Runtime.ExceptionServices.ExceptionDispatchInfo
    // 进行保留
    try
    {
        try
        {
            await SteveCreateSomeCodeAsync();
        }
        catch (DefectCreatedException dce)
        {
            Console.WriteLine(dce.ToString());
            await WriteEventLogEntry("ManagerApplication", dce.Message,
                EventLogEntryType.Error);
            throw;
        }
    }
    catch(DefectCreatedException dce)
    {
        Console.WriteLine(dce.ToString());
    }
}

public async Task WriteEventLogEntryAsync(string source, string message,
    EventLogEntryType type)
{
    await Task.Factory.StartNew(() => EventLog.WriteEntry(source, message, type));
}

public async Task SteveCreateSomeCodeAsync()
{
    Random rnd = new Random();
    await Task.Delay(rnd.Next(100, 1000));
    throw new DefectCreatedException("Null Reference",42);
}

public async Task JayCreateSomeCodeAsync()
{
    Random rnd = new Random();
    await Task.Delay(rnd.Next(100, 1000));
    throw new DefectCreatedException("Ambiguous Match",2);
}
```

```
public async Task TomCreateSomeCodeAsync()
{
    Random rnd = new Random();
    await Task.Delay(rnd.Next(100, 1000));
    throw new DefectCreatedException("Quota Exceeded",11);
}
public async Task SethCreateSomeCodeAsync()
{
    Random rnd = new Random();
    await Task.Delay(rnd.Next(100, 1000));
    throw new DefectCreatedException("Out Of Memory", 8);
}
```

自定义的 `DefectCreatedException` 类展示在例 5-8 中。

例 5-8：缺陷跟踪

```
[Serializable]
public class DefectCreatedException : Exception
{
    #region Constructors
    // 正常的异常构造函数
    public DefectCreatedException() : base()
    {
    }

    public DefectCreatedException(string message) : base(message)
    {
    }

    public DefectCreatedException(string message, Exception innerException)
        : base(message, innerException)
    {
    }

    // 接受新参数的异常构造函数
    public DefectCreatedException(string defect, int line) : base(string.Empty)
    {
        this.Defect = defect;
        this.Line = line;
    }

    public DefectCreatedException(string defect, int line, Exception innerException)
        : base(string.Empty, innerException)
    {
        this.Defect = defect;
        this.Line = line;
    }


    // 序列化构造函数
    protected DefectCreatedException(SerializationInfo exceptionInfo,
        StreamingContext exceptionContext)
        : base(exceptionInfo, exceptionContext)
    {
```

```
    }
    #endregion // Constructors

    #region Properties
    public string Defect { get; }
    public int Line { get; }

    public override string Message =>
        $"A defect was introduced: ({this.Defect ?? "Unknown"} on line {this.Line})";
    #endregion // Properties

    #region Overridden methods
    // ToString方法
    public override string ToString() =>
        $"{Environment.NewLine}{this.ToFullDisplayString()}";

    // 用于序列化时捕获额外字段的信息
    [SecurityPermission(SecurityAction.LinkDemand,
        Flags = SecurityPermissionFlag.SerializationFormatter)]
    public override void GetObjectData(SerializationInfo info,
        StreamingContext context)
    {
        base.GetObjectData(info, context);
        info.AddValue("Defect", this.Defect);
        info.AddValue("Line", this.Line);
    }
    #endregion // Overridden methods

    public string ToBaseString() => (base.ToString());
}
```

5.15.4 参考

MSDN 文档中的“async”“await”“AggregateException”“ConfigureAwait”和“System.Runtime.ExceptionServices.ExceptionDispatchInfo”主题。

5.16 有选择地处理异常

5.16.1 问题

你想要只处理会因多种原因而引发的异常中的某个特定实例。

5.16.2 解决方案

使用异常过滤器来仅捕获想要处理的状态的异常。

举例来说，假设在调用一个数据库时，你想要以不同的方式来处理超时情况。如果是从 ASP.NET WebApi 调用，那么当发现数据库超时错误时，你也许想要返回一个 503 服务不可用的消息来表明服务正忙。

ProtectedCallTheDatabase 方法将 CallTheDatabase 方法封装在一个 try-catch 语句块中，然后添加了一个异常过滤器（使用 when 关键字）检查 DatabaseException 分配的 Number 属性设置为 -2 的情况。当一个 DatabaseException 的 Number 属性被设置为 -2 时，说明是一个超时（就像当前的 Microsoft 数据库提供的），我们将捕获该异常并处理它。如果没有将 Number 设置为 -2，我们将不会捕获异常，它将在调用堆栈中向上传播到 ProtectedCallTheDatabase 方法的调用方。

```
private void ProtectedCallTheDatabase(string problem)
{
    try
    {
        CaallTheDatabase(problem);
        Console.WriteLine("No error on database call");
    }
    catch (DatabaseException dex) when (dex.Number == -2) // 观察超时
    {
        Console.WriteLine(
            "DatabaseException catch caught a database exception: " +
           $"{dex.Message}");
    }
}
```

CallTheDatabase 方法模拟了调用数据库并遇到一个问题，代码如下所示。

```
private void CallTheDatabase(string problem)
{
    switch (problem)
    {
        case "timeout":
            throw new DatabaseException(
                "Timeout expired. The timeout period elapsed prior to " +
                "completion of the operation or the server is not " +
                "responding. (Microsoft SQL Server, Error: -2).")
            {
                Number = -2,
                Class = 11
            };
        case "loginfail":
            throw new DatabaseException("Login failed for user")
            {
                Number = 18456,
            };
    }
}
```

可以用例 5-9 中展示的三种方式来调用 ProtectedCallTheDatabase 方法。

例 5-9：测试异常过滤器

```
Console.WriteLine("Simulating database call timeout");
try
{
    ProtectedCallTheDatabase("timeout");
}
```

```
catch(Exception ex)
{
    Console.WriteLine($"Exception catch caught a database exception: {ex.Message}");
}
Console.WriteLine("");

Console.WriteLine("Simulating database call login failure");
try
{
    ProtectedCallTheDatabase("loginfail");
}
catch (Exception ex)
{
    Console.WriteLine($"Exception catch caught a database exception: {ex.Message}");
}
Console.WriteLine("");

Console.WriteLine("Simulating successful database call");
try
{
    ProtectedCallTheDatabase("noerror");
}
catch (Exception ex)
{
    Console.WriteLine($"Exception catch caught a database exception: {ex.Message}");
}
Console.WriteLine("");
```

运行输出显示在例 5-10 中。

例 5-10：异常过滤器测试的输出

```
Simulating database call timeout
DatabaseException catch caught a database exception: Timeout expired.

The timeout period elapsed prior to completion of the operation or the server
is not responding. (Microsoft SQL Server, Error: -2).

Simulating database call login failure
Exception catch caught a database exception: Login failed for user

Simulating successful database call
No error on database call
```

可以看到，超时是在 `ProtectedCallTheDatabase` 的 `catch` 语句块中捕获的，而登录失败直到在测试代码中返回到 `catch` 语句块之后才会被捕获到。

5.16.3 讨论

异常过滤器允许你有条件地评估 `catch` 语句块是否要捕获一个异常，这是非常强大的，并允许你比以往更细粒度地仅处理能够进行处理的异常。

使用异常过滤器的另一个优点是它不需要不断地捕获和重新引发异常。如果做得不正确，捕获和重新引发异常会影响异常的调用堆栈并且隐藏错误（更多细节可参考范例 5.1，即

5.1 节），而异常过滤器允许对异常进行检查甚至执行操作（如日志记录）且不会干扰异常的原始流程。为了做到不干扰，异常过滤器中执行的代码必须返回 `false`，以使异常继续正常传播。在异常过滤器中用于判定 `true` 或 `false` 的代码应保持在最小规模，因为这是在 `catch` 处理器中，适用于同样的规则。不要做可能导致其他异常和掩盖初始想要捕捉的原始错误条件的事情。

`DatabaseException` 的完整清单如例 5-11 中所示。

例 5-11：`DatabaseException` 类

```
[Serializable]
public class DatabaseException : DbException
{
    public DatabaseException(string message) : base(message) { }
    public byte Class { get; set; }
    public Guid ClientConnectionId { get; set; }
    [DesignerSerializationVisibility(DesignerSerializationVisibility.Content)]
    public SqlErrorCollection Errors { get; set; }
    public int LineNumber { get; set; }
    public int Number { get; set; }
    public string Procedure { get; set; }
    public string Server { get; set; }
    public override string Source => base.Source;
    public byte State { get; set; }
    public override void GetObjectData(SerializationInfo si,
        StreamingContext context)
    {
        base.GetObjectData(si, context);
    }
}
```

5.16.4 参考

MSDN 文档中的“Exception Filters”主题。

第 6 章

反射和动态编程

6.0 简介

反射（reflection）是 .NET Framework 提供的一种机制，允许开发人员审视一个应用是如何构造的。使用反射，可以获得诸如程序集名称以及一个给定程序集导入的其他程序集等信息。开发人员甚至可以动态地调用给定程序集内一个类型实例的方法。反射还允许用户动态创建代码并编译成一个驻留内存的程序集或者构建一个程序集内的类型记录的符号表。

反射是 Framework 的一项非常强大的特性，受到运行时的保护。`ReflectionPermission` 必须被授予准备访问某一类型的保护或私有成员的程序集。如果只准备访问一个公共类型的公共成员，那就不需要被授予 `ReflectionPermission`。代码访问安全性（code access security，CAS）只有两个权限集默认授予所有反射访问，即 `FullTrust` 和 `Everything`。`LocalIntranet` 权限集包含 `ReflectionEmit` 特权（允许发布元数据并生成数据集）和 `MemberAccess` 特权（允许对程序集中类型的方法执行动态调用）。

本章中，你将会知道如何使用反射动态地调用类型上的成员，计算出一个给定程序集所依赖的所有程序集，并且审视程序集的不同类型的信息。反射是理解各个元素如何被集成到 .NET 中的一种好方法，而本章提供了学习起点。

本章还涵盖了 C# 中的 `dynamic` 关键字，它由 .NET 中的动态语言运行时（dynamic language runtime，DLR）提供支持。它用来帮助扩展 C# 在运行时识别一个对象的类型，而不是编译时的静态类型，以支持动态行为。要使用这些功能，需要引用 `System.Dynamic` 程序集和命名空间。

引入 DLR 是为了支持以下几个用例。

- 将其他语言（如 Python 和 Ruby）移植到 .NET

- 支持静态语言中的动态特性（如 C# 和 Visual Basic）
- 使语言之间共享更多库
- 缓存绑定操作（如反射）以提高性能，而不是运行时每次都确定一切

DLR 提供了以下三种主要服务。

- 表达式树（用以表示语言语义，如在 LINQ 中使用的那些）
- 调用站点缓存（在第一次执行时缓存操作特性）
- 动态对象互操作性（通过使用 IDynamicMetaObjectProvider、DynamicMetaObject、DynamicObject 和 ExpandoObject 实现）

C# 中提供的用于动态编程的三种主要构造是 dynamic 类型（一个未被编译时检查绑定的对象）、ExpandoObject 类（用于运行时构造或析构一个对象的成员）和 DynamicObject 类（用于将动态行为添加到自定义对象的基类）。本章论述了全部三个构造。

6.1 列出引用的程序集

6.1.1 问题

你需要确定某一特定程序集导入的所有程序集。该信息可以显示该程序集是否正在使用一个或多个你的程序集，或者它是否在使用另一个指定的程序集。

6.1.2 解决方案

使用 Assembly.GetReferencedAssemblies 方法，获得一个特定程序集的导入程序集，如例 6-1 所示。

例 6-1：使用 Assembly.GetReferencedAssemblies 方法

```
public static void BuildDependentAssemblyList(string path,
    StringCollection assemblies)
{
    // 维护一个原始程序集需要的程序集列表
    if(assemblies == null)
        assemblies = new StringCollection();

    // 是否已遇到过这个程序集
    if(assemblies.Contains(path)==true)
        return;

    try
    {
        Assembly asm = null;

        // 在字符串中查找常见的目录分隔符
        // 以确定这是一个文件名还是路径
        if ((path.IndexOf(@"\", 0, path.Length, StringComparison.Ordinal) != -1) ||
            (path.IndexOf("/", 0, path.Length, StringComparison.Ordinal) != -1))
        {
```

```
            // 从路径中加载程序集
            asm = Assembly.LoadFrom(path);
        }
        else
        {
            // 尝试用于程序集名称
            asm = Assembly.Load(path);
        }

        // 将程序集添加到列表
        if (asm != null)
            assemblies.Add(path);

        // 获得引用的程序集
        AssemblyName[] imports = asm.GetReferencedAssemblies();

        // 迭代
        foreach (AssemblyName asmName in imports)
        {
            // 递归调用此程序集以获得它引用的新模块
            BuildDependentAssemblyList(asmName.FullName, assemblies);
        }
    }
    catch (FileLoadException fle)
    {
        // 跳过这个程序集
        Console.WriteLine(fle);
    }
}
```

该代码返回一个 `StringCollection`，包含原始程序集、所有导入程序集以及导入程序集的依赖程序集。

如果针对程序集 C:\CSharpRecipes\bin\Debug\CSharpRecipes.exe 运行该方法，将得到下列依赖树。

```
Assembly C:\CSharpRecipes\bin\Debug\CSharpRecipes.exe has a dependency tree of
these assemblies:

    C:\CSharpRecipes\bin\Debug\CSharpRecipes.exe
    mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
    System, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
    System.Configuration, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a
    System.Xml, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
    System.Data.SqlXml, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
    System.Security, Version=4.0.0.0, Culture=neutral,
    PublicKeyToken=b03f5f7f11d50a3a
    System.Core, Version=4.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089
    System.Numerics, Version=4.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089
    System.Messaging, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a
```

```
    System.DirectoryServices, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a
    System.Transactions, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
    System.EnterpriseServices, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a
    System.Runtime.Remoting, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
    System.Web, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
    System.Drawing, Version=4.0.0.0, Culture=neutral,
    PublicKeyToken=b03f5f7f11d50a3a
    System.Data, Version=4.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089
    System.Web.RegularExpressions, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a
    System.Design, Version=4.0.0.0, Culture=neutral,
    PublicKeyToken=b03f5f7f11d50a3a
    System.Windows.Forms, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
    Accessibility, Version=4.0.0.0, Culture=neutral,
    PublicKeyToken=b03f5f7f11d50a3a
    System.Runtime.Serialization.Formatters.Soap, Version=4.0.0.0,
    Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a
    System.Deployment, Version=4.0.0.0, Culture=neutral,
    PublicKeyToken=b03f5f7f11d50a3a
    System.Data.OracleClient, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
    System.Drawing.Design, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a
    System.Web.ApplicationServices, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35
    System.ComponentModel.DataAnnotations, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35
    System.DirectoryServices.Protocols, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a
    System.Runtime.Caching, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a
    System.ServiceProcess, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a
    System.Configuration.Install, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a
    System.Runtime.Serialization, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
    System.ServiceModel.Internals, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35
    SMDiagnostics, Version=4.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089
    System.Web.Services, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a
    Microsoft.Build.Utilities.v4.0, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a
    Microsoft.Build.Framework, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a
    System.Xaml, Version=4.0.0.0, Culture=neutral,
```

```
    PublicKeyToken=b77a5c561934e089
    Microsoft.Build.Tasks.v4.0, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a
    NorthwindLinq2Sql, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=fe85c3941fbcc4c5
    System.Data.Linq, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
    System.Xml.Linq, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
    EntityFramework, Version=6.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
    Microsoft.CSharp, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a
    System.Dynamic, Version=4.0.0.0, Culture=neutral,
    PublicKeyToken=b03f5f7f11d50a3a
    System.Data.DataSetExtensions, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
```

6.1.3 讨论

获得一个程序集中的导入类型对于确定另一程序集正在使用哪些程序集非常有用。这一知识对学习使用一个新程序集很有帮助。该方法还有助于在发布前确定程序集之间的依赖性，或者在你被限制不能使用或导出程序集中的特定类型时执行合规管理。

System.Reflection.Assembly 类的 GetReferencedAssemblies 方法获得所有导入程序集的列表。该方法不接受任何参数，会返回一个 AssemblyName 对象数组而不是一个 Type 数组。AssemblyName 类型由名称、版本、区域信息、公钥 / 私钥对以及其他数据组成，这些成员允许你访问对应程序集的信息。

若要对当前的可执行文件调用 BuildDependentAssemblyList 方法，可运行下面的代码示例。

```
string file = GetProcessPath();

StringCollection assemblies = new StringCollection();

ReflectionAndDynamicProgramming.BuildDependentAssemblyList(file,assemblies);

Console.WriteLine($"Assembly {file} has a dependency tree of these
assemblies:{Environment.NewLine}");
foreach(string name in assemblies)
{
    Console.WriteLine($"\t{name}{Environment.NewLine}");
}
```

此处展示的 GetProcessPath 返回进程可执行文件的当前路径。

```
private static string GetProcessPath()
{
    // 修正路径,以便在调试器中运行时返回原始文件
    string processName = Process.GetCurrentProcess().MainModule.FileName;
    int index = processName.IndexOf("vshost", StringComparison.Ordinal);
    if (index != -1)
    {
```

```
            string first = processName.Substring(0, index);
            int numChars = processName.Length - (index + 7);
            string second = processName.Substring(index + 7, numChars);

            processName = first + second;
        }
        return processName;
    }
```

注意这个方法无法列出通过 `Assembly.ReflectionOnlyLoad*` 加载的程序集，因为它只审视编译时的引用。

> 将反射加载用于审视的程序集时，应当使用 `ReflectionOnlyLoad*` 方法。这些方法不允许执行来自加载程序集的代码。原因是你可能不知道自己是否正在加载包含了恶意代码的程序集。这些方法可防止所有恶意代码的执行。

6.1.4 参考

MSDN 文档中的“`Assembly` 类”主题。

6.2 确定程序集中的类型特征

6.2.1 问题

你需要找到一个程序集中具有某些特征的类型，例如以下这些。

- 通过方法名称
- 在程序集外部可用的类型
- 可序列化类型
- 给定类型的子类
- 嵌套类型

6.2.2 解决方案

使用反射来枚举与你正在寻找的特征匹配的类型。对于我们列出提纲的特征，可以使用表 6-1 中列出的方法。

表6-1：根据特征查找类型

特　　征	反射方法
方法名	`Type.GetMember`
导出类型	`Assembly.GetExportedTypes()`
可序列化类型	`Type.IsSerializeable`
类型的子类	`Type.IsSubclassOf`
嵌套类型	`Type.GetNestedTypes`

要在一个程序集中按名称查找方法，可使用扩展方法 GetMembersInAssembly，代码如下所示。

```
public static IEnumerable<MemberInfo> GetMembersInAssembly(this Assembly asm,
    string memberName) =>
    from type in asm.GetTypes()
        from ms in type.GetMember(memberName, MemberTypes.All,
            BindingFlags.Public | BindingFlags.NonPublic |
            BindingFlags.Static | BindingFlags.Instance)
        select ms;
```

GetMembersInAssembly 使用 Type.GetMember 来搜索具有匹配名称的所有成员并返回 MethodInfo 的集合。

```
var members = asm.GetMembersInAssembly(memberSearchName);
```

对于在程序集外部可用的类型，可使用 Assembly.GetExportedTypes 获得一个程序集的导出类型列表，代码如下所示。

```
var types = asm.GetExportedTypes();
```

要确定一个程序集中的 Serializable 类型，可使用扩展方法 GetSerializableTypes，代码如下所示。

```
public static IEnumerable<Type> GetSerializableTypes(this Assembly asm) =>
    from type in asm.GetTypes()
    where type.IsSerializable &&
        !type.IsNestedPrivate //  过滤掉匿名类型
    select type;
```

GetSerializableType 使用 Type.IsSerializable 属性确定类型是否支持序列化并返回一组可序列化类型。不用测试每个类型实现的接口和特性，查询 Type.IsSerializable 属性就可以确定它是否被标记为可序列化。

```
var serializeableTypes = asm.GetSerializableTypes();
```

要获得一个程序集中子类化某个特定类型的类型集，可使用扩展方法 GetSubclasses-ForType，代码如下所示。

```
public static IEnumerable<Type> GetSubclassesForType(this Assembly asm,
                                                     Type baseClassType) =>
    from type in asm.GetTypes()
    where type.IsSubclassOf(baseClassType)
    select type;
```

GetSubclassesForType 使用 Type.IsSubclassOf 方法确定一个程序集有哪些类型子类化了指定类型，它接受一个程序集路径字符串和代表基类的一个类型。该方法返回一个 IEnumerable<Type>，代表传递给 baseClassType 参数的类型的子类。在示例中，首先从当前进程中获得程序集路径，然后将 CSharpRecipes.ReflectionUtils+BaseOverrides 设置为测试子类的类型。调用 GetSubClassesForType，并返回一个 IEnumerable<Type>，代码如下所示。

```
Type type = Type.GetType(
    "CSharpRecipes.ReflectionAndDynamicProgramming+BaseOverrides");
var subClasses = asm.GetSubclassesForType(type);
```

最后，要确定程序集中的嵌套类型，可使用扩展方法 GetNestedTypes，代码如下所示。

```
public static IEnumerable<Type> GetNestedTypes(this Assembly asm) =>
    from t in asm.GetTypes()
        from t2 in t.GetNestedTypes(BindingFlags.Instance |
                    BindingFlags.Static |
                    BindingFlags.Public |
                    BindingFlags.NonPublic)
        where !t2.IsEnum && !t2.IsInterface &&
              !t2.IsNestedPrivate // 过滤掉匿名类型
        select t2;
```

GetNestedTypes 使用 Type.GetNestedTypes 方法，并审视程序集中的每个类型以确定它是否包含嵌套，代码如下所示。

```
var nestedTypes = asm.GetNestedTypes();
```

6.2.3 讨论

你为什么应该关心这些关于程序集中类型的随机事实呢？因为它们能帮助你弄清楚代码是如何构建的，发现你可能会允许或不会允许的编码实践。让我们单独看看每一项，以便知道为什么你也许想要了解它。

1. 方法名

memberName 参数可以包含通配符 *，表示任意单个或多个字符。因此，要查找以字符串 "Test" 开头的所有方法，可以将字符串 "Test*" 传递给 memberName 参数。要注意，memberName 参数是区分大小写的，但 asmPath 参数不是。如果你想要不区分大小写的成员查找，可将 BindingFlags.IgnoreCase 标志添加到 Type.GetMember 调用的其他 BindingFlags 中。

System.Type 类的 GetMember 方法对于查找一个类型中的一个或多个方法非常有用。该方法返回一个 MemberInfo 对象数组，描述了匹配给定参数的所有成员。

> * 字符只有在 *name* 参数字符串末尾才用作通配符。如果放置在字符串中的其他位置，则不会将其视为通配符。此外，为确保返回所有成员，* 必须是 *name* 参数中的唯一字符。诸如 ? 之类的其他通配符都不受支持。

获得了一个 MemberInfo 对象的数组之后，你需要检查这些成员的类型。MemberInfo 类包含一个 MemberType 属性，返回一个 System.Reflection.MemberTypes 枚举值，它可以是表 6-2 中定义的 All 之外的任何值。

表6-2：MemberTypes枚举值

枚举值	定　义
All	所有成员类型
Constructor	一个构造函数成员

（续）

枚举值	定　义
Custom	一个自定义成员类型
Event	一个事件成员
Field	一个字段成员
Method	一个方法成员
NestedType	一个嵌套成员
TypeInfo	一个代表 TypeInfo 成员的类型成员

2. 导出类型

获得一个程序集中的导出类型在确定该程序集的公共接口时非常有用。这种能力非常有助于学习使用一个新的程序集，或者可以帮助程序集的开发人员确定该程序集的所有访问点，以检验它们对于恶意代码足够安全。要获得这些导出类型，可使用 System.Reflection.Assembly 类型上的 GetExportedTypes 方法。导出的类型由从该程序集外部可公共访问的所有类型构成。一个类型可能具有公共访问性，但从程序集外部不可访问。以下列代码为例。

```
public class Outer
{
    public class Inner {}
    private class SecretInner {}
}
```

导出的类型是 Outer 和 Outer.Inner，类型 SecretInner 未被导出到该程序集的外部。如果将 Out 的可达性由 public 修改为 private，就没有了能够外部访问的类型，因为 Outer 类上的 private，所以 Inner 类访问级别降低。

3. 可序列化类型

使用 SerializableAttribute 特性可将一个类型标记为可序列化的。测试一个类型上的 SerializableAttribute 属性将会导致大量的工作。这是因为 SerializableAttribute 是一种不可思议的特性，C# 编译器在编译时实际上去掉了标记代码。使用 ildasm（.NET 平台的反编译器），你将看到这一自定义特性并不存在，通常会看到针对每个自定义特性的一个 .custom 记录，但 SerializableAttribute 不是。C# 编译器移除了它，取而代之的是，在类的元数据中设置了一个标志。在源代码中，它看起来好像一个自定义特性，但它编译为一个小特性集中的一个特性，在元数据中有特殊表示。这就是它在反射 API 中得到特殊对待的原因。幸运的是，你不必完成所有的工作。如果使用 SerializableAttribute 将当前类型标记为可序列化的，那么 Type 类型上的 IsSerializable 属性返回 true，否则该属性返回 false。

4. 类型的子类

Type 类上的 IsSubclassOf 方法允许你确定当前类型是否传递给该方法的类型的一个子类。通过了解一个类型是否被子类化，可以探索团队或公司创建的类型层次结构，从而带来代码重用、重构或更好理解代码库中依赖的机会。

5. 嵌套类型

通过确定嵌套类型，可以通过编程检查某些设计模式的各个方面。各种设计模式都可能指定一个类型将包含另一个类型。例如，修饰器设计模式和状态设计模式使用了对象包含。

`GetNestedTypes` 扩展方法使用一个 LINQ 查询来查询 `asmPath` 参数指定的程序集中的所有类型。LINQ 查询还使用了 `Type` 类的 `GetNestedTypes` 方法来查询程序集中的嵌套类型。

点运算符通常用于分隔命名空间和类型，但是嵌套类型有些特殊。在反射 API 中处理嵌套类型时，它们在其完全限定名中使用 + 与其他类型分开。通过将这一完全限定名传递给静态 `GetType` 方法，可以获得它所代表的实际类型。

这些方法返回一个代表 `typeName` 参数标识的类型的 `Type` 对象。

> 调用 `Type.GetType` 以获得在动态程序集（使用在 `System.Reflection.Emit` 命名空间中定义的类型所生成的程序集）中定义的一个类型，如果该程序集尚未持久化到磁盘，那么会返回一个 `null`。通常应该使用动态程序集 `Assembly` 对象和静态 `Assembly.GetType` 方法。

6.2.4 参考

MSDN 文档中的“`Assembly` 类”“`Type` 类”“`TypeAttributes` 枚举”“`BindingFlags` 枚举”和“`MemberInfo` 类”主题。

6.3 确定继承特征

6.3.1 问题

你需要确定类型的以下两个继承特征。

- 继承层次结构
- 被重写的基类方法

6.3.2 解决方案

使用反射来枚举继承链和基类方法重写，如表 6-3 所示。

表6-3：根据特征查找类型

特　　征	反射方法
继承层次结构	`Type.BaseType`
基类方法	`MethodInfo.GetBaseDefinition`

使用扩展方法 `GetInheritanceChain` 来获得单个类型的整个继承层次结构。`GetInheritanceChain` 使用 `GetBaseTypes` 方法枚举类型，然后将默认顺序反转以从基类到派生类的排序顺序提供枚举列表。换句话说，当 `GetBaseTypes` 遍历每个遇到的类型

的 BaseType 时，类型列表的结果是从最大派生深度到最小派生深度的顺序，因此调用 Reverse 以将列表排序为最小派生深度的类型（Object）在最前面，代码如下所示。

```
public static IEnumerable<Type> GetInheritanceChain(this Type derivedType) =>
    (from t in derivedType.GetBaseTypes()
    select t).Reverse();

private static IEnumerable<Type> GetBaseTypes(this Type type)
{
    Type current = type;
    while (current != null)
    {
        yield return current;
        current = current.BaseType;
    }
}
```

如果你想要对程序集中的所有类型执行此操作，可以使用扩展方法 GetTypeHierarchies。它使用自定义的 TypeHierarchy 类来表示派生类型及其继承链。

```
public class TypeHierarchy
{
    public Type DerivedType { get; set; }
    public IEnumerable<Type> InheritanceChain { get; set; }
}

public static IEnumerable<TypeHierarchy> GetTypeHierarchies(this Assembly asm) =>
    from Type type in asm.GetTypes()
    select new TypeHierarchy
    {
        DerivedType = type,
        InheritanceChain = GetInheritanceChain(type)
    };
```

GetTypeHierarchies 将每个类型表现为 DerivedType，并使用 GetInheritanceChain 来确定类型的 InheritanceChain。

要确定基类方法是否被重写，可使用 MethodInfo.GetBaseDefinition 方法来确定基类中的哪个方法被重写了。例 6-2 中所示的扩展方法 GetMethodOverrides 检查类中所有的公开实例方法并显示哪些方法重写了它们各自的基类方法。该方法还确定被重写的方法位于哪个基类。此扩展方法基于 Type 并使用类型查找重写的方法。

例 6-2：GetMethodOverrides 方法

```
public class ReflectionUtils
{
    public static IEnumerable<MemberInfo> GetMethodOverrides(this Type type) =>
        from ms in type.GetMethods(BindingFlags.Instance |
                                   BindingFlags.NonPublic | BindingFlags.Public |
                                   BindingFlags.Static | BindingFlags.DeclaredOnly)
        where ms != ms.GetBaseDefinition()
        select ms.GetBaseDefinition;
}
```

下一个扩展方法 GetBaseMethodOverridden 使你可以确定是否某个特定的方法重写了其基类中的方法，并返回与被重写方法对应的 MethodInfo。它同样扩展了 Type，参数为完整的方法名称和表示其参数类型的 Type 对象数组，代码如下所示。

```
public class ReflectionUtils
{
    public static MethodInfo GetBaseMethodOverridden(this Type type,
                                        string methodName, Type[] paramTypes)
    {
        MethodInfo method = type.GetMethod(methodName, paramTypes);
        MethodInfo baseDef = method?.GetBaseDefinition();
        if (baseDef != method)
        {
            bool foundMatch = (from p in baseDef.GetParameters()
                        join op in paramTypes
                            on p.ParameterType.UnderlyingSystemType
                                equals op.UnderlyingSystemType
                        select p).Any();

            if (foundMatch)
                return baseDef;
        }
        return null;
    }
}
```

6.3.3　讨论

1. 继承层次结构

不幸的是，不存在 Type 类的一个属性可以获得一个类型的继承层次结构。不过，本范例中的 GetInheritanceChain 方法可以做到这一点，只需要传入要获取继承层次结构的类型的 type 参数。GetTypeHierarchies 只需要一个程序集参数，因为它生成了程序集中所有类型的继承层次结构。

本范例的核心代码位于 GetBaseTypes 方法。这是一个枚举器方法，它遍历每个继承的类型，直至找到最终的基类——它总会是一个 object 类。一旦它到达这一最终基类，便会停止迭代。GetBaseTypes 返回的 IEnumerable<Type> 包含了所有基类。

要显示某个类型的继承链，可以使用 DisplayInheritanceChain 方法调用。

```
private static void DisplayInheritanceChain(IEnumerable<Type> chain)
{
    StringBuilder builder = new StringBuilder();
    foreach (var type in chain)
    {
        if (builder.Length == 0)
            builder.Append(type.Name);
        else
            builder.AppendFormat($"<-{type.Name}");
    }
    Console.WriteLine($"Base Type List: {builder.ToString()}");
}
```

要显示一个程序集中所有类型的继承层次结构，可以结合使用 GetTypeHierarchies 和 DisplayInheritanceChain，代码如下所示。

```
// 程序集中的所有类型
var typeHierarchies = asm.GetTypeHierarchies();
foreach (var th in typeHierarchies)
{
    // 递归所有基类
    Console.WriteLine($"Derived Type: {th.DerivedType.FullName}");
    DisplayInheritanceChain(th.InheritanceChain);
    Console.WriteLine();
}
```

这些方法产生如下输出。

```
Derived Type: CSharpRecipes.Reflection
Base Type List: Object<-Reflection
Derived Type: CSharpRecipes.ReflectionUtils+BaseOverrides
Base Type List: Object<-BaseOverrides

Derived Type: CSharpRecipes.ReflectionUtils+DerivedOverrides
Base Type List: Object<-BaseOverrides <-DerivedOverrides
```

该输出显示，当查看 CSharpRecipes 命名空间中的 Reflection 类时，其基类型列表（或称作继承层次结构）以 Object 开头（与 .NET 中的所有类和结构类似）。嵌套类 BaseOverrides 还显示一个以 Object 开头的基类型列表。嵌套类 DerivedOverrides 显示一个更有趣一些的基类型列表，其中 DerivedOverrides 从 BaseOverrides 派生，而 BaseOverrides 从 Object 派生。

2. 被重写的基类方法

如果没有 System.Reflection.MethodInfo 类型的 GetBaseDefinition 方法，那么确定哪些方法重写了其基类方法是一件非常烦琐的事情。该方法没有参数，返回一个对应于基类中被重写方法的 MethodInfo 对象。如果该方法被用于一个代表未被重写方法的 MethodInfo 对象（与用于某个虚或抽象方法的情况相同），那么 GetBaseDefinition 返回原始的 MethodInfo 对象。

当方法名及其参数数组都被传入 GetBaseMethodOverridden 时，会调用 Type 对象的 GetMethod 方法，否则会将 GetMethods 用于 GetMethodOverrides。如果正确地定位了方法并获得了其 MethodInfo 对象，则会在 MethodInfo 对象上调用 GetBaseDefinition 方法，以获得继承层次中最近基类中的第一个被重写的方法。将该 MethodInfo 的类型与调用了 GetBaseDefinition 方法的 MethodInfo 的类型进行比较。如果两个对象相同，那么这意味着所有基类中都没有被重写的方法；因此，不会返回任何内容。该方法只会返回被重写的方法；如果没有方法被重写，则返回 null。

下列代码展示了如何使用这些重载方法。

```
Type derivedType =
  asm.GetType("CSharpRecipes.ReflectionAndDynamicProgramming+DerivedOverrides",
    true, true);
```

```
var methodOverrides = derivedType.GetMethodOverrides();
foreach (MethodInfo mi in methodOverrides)
{
    Console.WriteLine();
    Console.WriteLine($"Current Method: {mi.ToString()}");
    Console.WriteLine($"Base Type FullName:  {mi.DeclaringType.FullName}");
    Console.WriteLine($"Base Method:  {mi.ToString()}");
    // 列出此方法的类型
    foreach (ParameterInfo pi in mi.GetParameters())
    {
        Console.WriteLine($"\tParam {pi.Name} : {pi.ParameterType.ToString()}");
    }
}

// 寻找更多重载
string methodName = "Foo";
var baseTypeMethodInfo = derivedType.GetBaseMethodOverridden(methodName,
    new Type[3] { typeof(long), typeof(double), typeof(byte[]) });
Console.WriteLine(
    $"{Environment.NewLine}For [Type] Method: [{derivedType.Name}]" +
    $" {methodName}");
Console.WriteLine(
    $"Base Type FullName: {baseTypeMethodInfo.ReflectedType.FullName}");
Console.WriteLine($"Base Method: {baseTypeMethodInfo}");
foreach (ParameterInfo pi in baseTypeMethodInfo.GetParameters())
{
    // 列出参数以便了解取得了哪一个
    Console.WriteLine($"\tParam {pi.Name} : {pi.ParameterType.ToString()}");
}
```

在使用代码中，通过 Process 类获得指向测试代码程序集（CSharpRecipes.exe）的路径。然后使用它查找在 ReflectionUtils 类中定义的 DerivedOverrides 类，它从 BaseOverrides 类派生。DerivedOverrides 和 BaseOverrides 如下所示。

```
public abstract class BaseOverrides
{
    public abstract void Foo(string str, int i);

    public abstract void Foo(long l, double d, byte[] bytes);
}

public class DerivedOverrides : BaseOverrides
{
    public override void Foo(string str, int i)
    {
    }

    public override void Foo(long l, double d, byte[] bytes)
    {
    }
}
```

GetMethodOverrides 返回它在 Reflection.DerivedOverrides 类型中找出的每个方法的所有被重写的方法。如果希望显示所有重写方法及其相应被重写的方法，那么可以从

GetMethods 方法调用中移除 BindingFlags.DeclaredOnly 绑定枚举，代码如下所示。

```
return from ms in type.GetMethods(BindingFlags.Instance |
            BindingFlags.NonPublic | BindingFlags.Public)
        where ms != ms.GetBaseDefinition()
        select ms.GetBaseDefinition();
```

GetBaseMethodOverridden 传入一个方法名以及该方法的参数，以根据这些参数查找匹配签名的重写版本。在本例中，方法 Foo 的参数类型是 long、double 和 byte[]。该方法显示 DerivedOverrides.Foo 重写的方法。

6.3.4　参考

MSDN 文档中的“Assembly 类”“Type.BaseType 方法”“MethodInfo 类”和“ParameterInfo 类”主题。

6.4　使用反射调用成员

6.4.1　问题

你有一个方法名列表，希望在应用程序内动态地调用它们。当代码运行时，将名称从列表中取出并尝试调用这些方法。这一技术对于创建组件测试工具非常有用，可以用于从一个 XML（或 JSON）文件中读入方法并用给定的参数执行。

6.4.2　解决方案

例 6-3 所示的 TestReflectionInvocation 方法调用 ReflectionInvoke 方法，后者打开 XML 配置文件，使用 LINQ 读入测试信息并执行每个测试方法。

例 6-3：通过反射调用成员

```
public static void TestReflectionInvocation()
{
    XDocument xdoc =
        XDocument.Load(@"..\..\SampleClassLibrary\SampleClassLibraryTests.xml");
    ReflectionInvoke(xdoc, @"SampleClassLibrary.dll");
}
```

测试方法信息所在的 XML 文档如下所示。

```
<?xml version="1.0" encoding="utf-8" ?>
<Tests>
    <Test className='SampleClassLibrary.SampleClass'
    methodName='TestMethod1'>
        <Argument>Running TestMethod1</Argument>
    </Test>
    <Test className='SampleClassLibrary.SampleClass'
    methodName='TestMethod2'>
        <Parameter>Running TestMethod2</Parameter>
        <Parameter>27</Parameter>
```

```
    </Test>
</Tests>
```

如例 6-4 中所示，ReflectionInvoke 使用 XDocument 中包含的信息动态地调用传递给它的方法。每个参数的类型通过检查 MethodInfo 上的 ParameterInfo 数据项来确定，然后通过 Convert.Changetype 方法将提供的值从一个字符串转换为实际类型。最终，被调用方法的返回值由 MethodBase.Invoke 方法返回。

例 6-4：ReflectionInvoke 方法

```
public static void ReflectionInvoke(XDocument xdoc, string asmPath)
{
    var test = from t in xdoc.Root.Elements("Test")
               select new
               {
                   typeName = (string)t.Attribute("className").Value,
                   methodName = (string)t.Attribute("methodName").Value,
                   parameter = from p in t.Elements("Parameter")
                               select new { arg = p.Value }
               };

    // 加载程序集
    Assembly asm = Assembly.LoadFrom(asmPath);

    foreach (var elem in test)
    {
        // 创建实际类型
        Type reflClassType = asm.GetType(elem.typeName, true, false);

        // 创建此类型的一个实例并验证它是否存在
        object reflObj = Activator.CreateInstance(reflClassType);
        if (reflObj != null)
        {
            // 验证方法是否存在,并获取MethodInfo对象
            MethodInfo invokedMethod = reflClassType.GetMethod(elem.methodName);
            if (invokedMethod != null)
            {
                // 创建动态调用所需的参数列表
                object[] arguments = new object[elem.parameter.Count()];
                int index = 0;

                // 将每一个参数添加到列表中
                foreach (var arg in elem.parameter)
                {
                    // 获得参数类型
                    Type paramType =
                        invokedMethod.GetParameters()[index].ParameterType;

                    // 转换为对应类型并赋值
                    arguments[index] =
                        Convert.ChangeType(arg.arg, paramType);
                    index++;
                }

                // 使用参数调用方法
```

```
                    object retObj = invokedMethod.Invoke(reflObj, arguments);

                    Console.WriteLine($"\tReturned object: {retObj}");
                    Console.WriteLine($"\tReturned object: {retObj.GetType().FullName}");
                }
            }
        }
    }
```

以下这些是动态调用的方法，位于 SampleClassLibrary 程序集中的 SampleClass 类型上。

```
public bool TestMethod1(string text)
{
    Console.WriteLine(text);
    return (true);
}

public bool TestMethod2(string text, int n)
{
    Console.WriteLine(text + " invoked with {0}",n);
    return (true);
}
```

这些方法的输出如下所示。

```
Running TestMethod1
        Returned object: True
        Returned object: System.Boolean
Running TestMethod2 invoked with 27
        Returned object: True
        Returned object: System.Boolean
```

6.4.3 讨论

反射赋予用户动态调用相同程序集或不同程序集中某一类型内的静态方法和实例方法的能力。这是一种非常强大的工具，允许用户代码在运行时确定调用哪个方法。这一确定过程可基于程序集名称、类型名称或方法名称，即使在以下这些并不需要程序集名称的情况下也是如此：方法位于调用代码所处的同一个程序集中，已经获得了 Assembly 对象，或者获得了方法所在的类的 Type 对象。

> 像往常一样，能力越大、责任越大。动态加载程序集而不知道其来源（或者甚至是在一个提升的上下文中的合法调用）会造成不需要的后果，所以要明智、安全地使用这种技术。

这种技术看起来与委托类似，因为两者都能够在运行时动态地决定调用哪个方法。总的来说，委托要求用户知道在运行时可能调用的方法签名，而使用反射，用户可以在对签名一无所知的情况下调用方法，提供更宽松的绑定。但是，用户仍需要传递合理的参数。使用 Delegate.DynamicInvoke 可完成更加动态的调用，但它更大程度上是一个基于反射的方法而不是传统的委托调用。

6.4.2 节所示的 ReflectionInvoke 方法包含动态调用一个方法所需的所有代码。该代码首先使用程序集名加载程序集（通过 asmPath 参数传递），然后获得包含针对要调用方法的类的 Type 对象（类名使用 LINQ 从 Test 元素的 className 属性中获得），然后使用 LINQ 从 Test 元素的 methodName 属性中重新获得方法名。一旦从 Test 元素中获得了所有信息，生成一个 Type 对象的实例，然后就可以调用这个生成的实例上的指定方法，操作步骤如下所示。

- 首先，调用静态 Activator.CreateInstance 方法实际生成局部变量 reflClassType 中包含的 Type 对象的一个实例。方法返回一个指向生成的类型实例的对象引用，如果对象不能生成则引发一个异常。
- 用户成功获得该类的实例之后，通过调用 Type 对象上的 GetMethod 获得要调用方法的 MethodInfo 对象。

然后，将使用 CreateInstance 方法生成的对象实例作为第一个参数传递给 MethodInfo.Invoke 方法。该方法返回一个包含被调用方法的返回值的对象。之后，InvokeMethod 返回该对象。传递给 MethodInfo.Invoke 的第二个参数是一个包含所有要传递给该方法的参数的对象数组。该数组根据 XML 中每个 Test 元素节点下的 Parameter 元素数目进行构建。随后会看到每个参数的 ParameterInfo（通过 MethodInfo.GetParameters 获得），并使用 Convert.ChangeType 方法将 XML 中的字符串值转换为正确的类型。

ReflectionInvoke 方法最后显示每个返回的对象值及其类型。注意到从被调用的方法中返回不同的返回值不需要额外的逻辑，因为返回值都是作为一个对象返回的，与将不同参数传递给被调用的方法时不同。

6.4.4 参考

MSDN 文档中的“Activator 类”“MethodInfo 类”“Convert.ChangeType 方法”和“ParameterInfo 类”主题。

6.5 访问局部变量信息

6.5.1 问题

你正在构建一个检查代码的工具，需要访问一个方法内的局部变量。

6.5.2 解决方案

使用 MethodBody 类上的 LocalVariables 属性以返回一个 LocalVariableInfo 对象的 IList，其中每个对象描述了方法内的一个局部变量。

```
public static ReadOnlyCollection<LocalVariableInfo>
GetLocalVars(string asmPath, string typeName, string methodName)
{
    Assembly asm = Assembly.LoadFrom(asmPath);
    Type asmType = asm.GetType(typeName);
```

```
        MethodInfo mi = asmType.GetMethod(methodName);
        MethodBody mb = mi.GetMethodBody();

        System.Collections.ObjectModel.ReadOnlyCollection<LocalVariableInfo> vars =
            (System.Collections.ObjectModel.ReadOnlyCollection<LocalVariableInfo>)
                mb.LocalVariables;

        // 显示每个局部变量的信息
        foreach (LocalVariableInfo lvi in vars)
        {
            Console.WriteLine($"IsPinned: {lvi.IsPinned}");
            Console.WriteLine($"LocalIndex: {lvi.LocalIndex}");
            Console.WriteLine($"LocalType.Module: {lvi.LocalType.Module}");
            Console.WriteLine($"LocalType.FullName: {lvi.LocalType.FullName}");
            Console.WriteLine($"ToString(): {lvi.ToString()}");
        }

        return (vars);
    }
```

GetLocalVars 方法可使用下列代码进行调用。

```
    public static void TestGetLocalVars()
    {
        string file = GetProcessPath();

        // 获得CSharpRecipes.Reflection.GetLocalVars
        // 方法内的所有局部变量信息
        System.Collections.ObjectModel.ReadOnlyCollection<LocalVariableInfo> vars =
            GetLocalVars(file, "CSharpRecipes.ReflectionAndDynamicProgramming",
                "GetLocalVars");
    }
```

此处展示的 GetProcessPath 返回进程可执行文件的当前路径。

```
    private static string GetProcessPath()
    {
        // 修正路径,以便在调试器中运行时返回原始文件
        string processName = Process.GetCurrentProcess().MainModule.FileName;
        int index = processName.IndexOf("vshost", StringComparison.Ordinal);
        if (index != -1)
        {
            string first = processName.Substring(0, index);
            int numChars = processName.Length - (index + 7);
            string second = processName.Substring(index + 7, numChars);

            processName = first + second;
        }
        return processName;
    }
```

这个方法的输出如下所示。

```
    IsPinned: False
    LocalIndex: 0
    LocalType.Module: CommonLanguageRuntimeLibrary
```

```
LocalType.FullName: System.Reflection.Assembly
ToString(): System.Reflection.Assembly (0)
IsPinned: False
LocalIndex: 1
LocalType.Module: CommonLanguageRuntimeLibrary
LocalType.FullName: System.Type
ToString(): System.Type (1)
IsPinned: False
LocalIndex: 2
LocalType.Module: CommonLanguageRuntimeLibrary
LocalType.FullName: System.Reflection.MethodInfo
ToString(): System.Reflection.MethodInfo (2)
IsPinned: False
LocalIndex: 3
LocalType.Module: CommonLanguageRuntimeLibrary
LocalType.FullName: System.Reflection.MethodBody
ToString(): System.Reflection.MethodBody (3)
IsPinned: False
LocalIndex: 4
LocalType.Module: CommonLanguageRuntimeLibrary
LocalType.FullName: System.Collections.ObjectModel.ReadOnlyCollection`1[[System.
Reflection.LocalVariableInfo, mscorlib, Version=4.0.0.0, Culture=neutral, Public
KeyToken=b77a5c561934e089]]
ToString(): System.Collections.ObjectModel.ReadOnlyCollection`1[System.Reflectio
n.LocalVariableInfo] (4)
IsPinned: False
LocalIndex: 5
LocalType.Module: CommonLanguageRuntimeLibrary
LocalType.FullName: System.Collections.Generic.IEnumerator`1[[System.Reflection.
LocalVariableInfo, mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b7
7a5c561934e089]]
ToString(): System.Collections.Generic.IEnumerator`1[System.Reflection.LocalVari
ableInfo] (5)
IsPinned: False
LocalIndex: 6
LocalType.Module: CommonLanguageRuntimeLibrary
LocalType.FullName: System.Reflection.LocalVariableInfo
ToString(): System.Reflection.LocalVariableInfo (6)
IsPinned: False
LocalIndex: 7
LocalType.Module: CommonLanguageRuntimeLibrary
LocalType.FullName: System.Collections.ObjectModel.ReadOnlyCollection`1[[System.
Reflection.LocalVariableInfo, mscorlib, Version=4.0.0.0, Culture=neutral, Public
KeyToken=b77a5c561934e089]]
ToString(): System.Collections.ObjectModel.ReadOnlyCollection`1[System.Reflectio
n.LocalVariableInfo] (7)
```

在 CSharpRecipes.Reflection.GetLocalVars 方法中找到的、用于每个局部变量的 LocalVariableInfo 对象将在 vars IList 集合中返回。

6.5.3 讨论

LocalVariables 属性可给出关于一个方法内变量的大量信息，它返回一个 IList<LocalVariableInfo> 集合。每个 LocalVariableInfo 对象都包含表 6-4 中描述的信息。

表6-4：LocalVariableInfo信息

成　员	描　述
IsPinned	返回一个表示这个变量引用的对象是（true）否（false）被固定在内存中的布尔值。在非托管代码中，一个对象必须首先被固定才能被非托管指针引用。当对象被固定时，不能被垃圾回收移动
LocalIndex	返回方法体内这个变量的索引
LocalType	返回一个描述这个变量类型的 Type 对象
ToString	返回 LocalType.FullName、一个空格，然后是用括号括住的 LocalIndex 值

6.5.4　参考

MSDN 文档中的“MethodInfo 类”“MethodBody 类”“ReadOnlyCollection<T> 类”和“LocalVariableInfo 类”主题。

6.6　创建一个泛型类型

6.6.1　问题

你想要只使用反射 API 来生成一个泛型类型。

6.6.2　解决方案

创建一个泛型类型类似于创建一个非泛型类型，但是在构建时需要一个额外的步骤创建想要使用的类型实参并且将这些类型实参绑定到泛型类型的类型形参。为此，将要使用一个添加到 Type 类上的新方法，名为 BindGenericParameters。

```
public static void CreateDictionary()
{
    // 获得想要构造的类型
    Type typeToConstruct = typeof(Dictionary<,>);
    // 获得想要使用的类型参数
    Type[] typeArguments = {typeof(int), typeof(string)};
    // 将类型参数绑定到泛型类型
    Type newType = typeToConstruct.MakeGenericType(typeArguments);

    // 构造泛型类型
    Dictionary<int, string> dict =
        (Dictionary<int, string>)Activator.CreateInstance(newType);

    // 测试新构造的类型
    Console.WriteLine($"Count == {dict.Count}");
    dict.Add(1, "test1");
    Console.WriteLine($"Count == {dict.Count}");
}
```

测试 CreateDictionary 方法的代码如下所示。

```
public static void TestCreateMultiMap()
{
    Assembly asm = Assembly.LoadFrom("C:\\CSCB6 " +
             "\\Code\\CSharpRecipes\\bin\\Debug\\CSharpRecipes.exe");
    CreateDictionary(asm);
}
```

这一方法的输出如下所示。

```
Count == 0
Count == 1
```

6.6.3 讨论

类型形参在一个类上定义，表示所有能够被转换为 `Object` 的类型都允许替代这一类型形参（当然，除非在这一类型形参上存在使用 `where` 关键字添加的约束）。例如，下列类有两个类型形参 `T` 和 `U`。

```
public class Foo<T, U> {...}
```

> 当然，你并非一定要使用 `T` 和 `U`，还可以使用其他字母甚至是完整的名称，如 `TypeParam1` 和 `TypeParam2`。

一个类型实参被定义为将要替代类型形参的实际类型。在前面定义的类 `Foo` 中，用户可以用类型实参 `int` 替代类型形参 `T`，用类型实参 `string` 替代类型形参 `U`。

`BindGenericParameters` 方法允许用户用实际类型实参替代类型形参。这一方法接受一个 `Type` 数组参数。这个 `Type` 数组由将替代泛型类型的每个类型形参的类型实参构成。这些类型实参必须按它们在类中定义的顺序添加到这个 `Type` 数组中。例如，`Foo` 类按顺序定义了类型形参 `T` 和 `U`。开发人员定义的 `Type` 数组按这个顺序包含一个 `int` 类型和一个 `string` 类型。这意味着，类型形参 `T` 将被类型实参 `int` 替代，而 `U` 将被一个 `string` 类型替代。`BindGenericParameters` 方法将返回一个使用指定类型实参的类型的 `Type` 对象。

6.6.4 参考

MSDN 文档中的“`Type.BindGenericParameters` 方法”主题。

6.7 使用dynamic与使用object

6.7.1 问题

你想要知道使用 `dynamic` 和 `object` 作为类型规范时的不同点。

6.7.2 解决方案

要演示 dynamic 和 object 之间的主要区别，我们将回顾范例 6.4（即 6.4 节）中使用的示例类。那段代码动态地加载 SampleClass 类型的一个实例，然后使用一个 XML 文件和反射机制，运行实例上的某些操作。那个实例是 object 类型。如果我们创建类型并将其标记为 dynamic，就可以编写代码以在代码中直接调用方法（放弃了前一示例中的灵活性，但是使代码更简洁），即使 dynamic 对象实例并不是 SampleClass 类型也是如此。

```
// 加载程序集
Assembly asm = Assembly.LoadFrom(@"SampleClassLibrary.dll");

// 获得SampleClass类型
Type reflClassType = asm?.GetType("SampleClassLibrary.SampleClass", true, false);

if (reflClassType != null)
{
    // 创建示例类实例
    dynamic sampleClass = Activator.CreateInstance(reflClassType);
    Console.WriteLine($"LastMessage: {sampleClass.LastMessage}");
    Console.WriteLine("Calling TestMethod1");
    sampleClass.TestMethod1("Running TestMethod1");
    Console.WriteLine($"LastMessage: {sampleClass.LastMessage}");
    Console.WriteLine("Calling TestMethod2");
    sampleClass.TestMethod2("Running TestMethod2", 27);
    Console.WriteLine($"LastMessage: {sampleClass.LastMessage}");
}
```

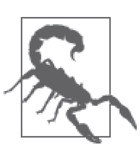
注意到我们可以直接调用方法而不会产生错误，即使对象实例的类型是 dynamic。这是因为编译器知道要推迟这些调用（LastMessage、TestMethod1、TestMethod2）的类型检查直到运行时。尽管对待 dynamic 的方式像对待 object 一样，甚至 dynamic 最终编译为 object，但是它会告诉编译器："嘿，放松些，我知道自己在做什么！"，并允许你调用编译器无法解析的方法和属性。

这一示例的输出如下所示。

```
LastMessage: Not set yet
Calling TestMethod1
Running TestMethod1
LastMessage: Running TestMethod1
Calling TestMethod2
Running TestMethod2 invoked with 27
LastMessage: Running TestMethod2
```

6.7.3 讨论

dynamic 类型允许你绕过编译时类型检查，并在运行时将操作绑定到调用站点。



请记住，如果访问了某个动态对象上不存在的成员，而且在运行之前没有察觉的话，将会遇到未预期的异常。

大多数情况下，dynamic 的行为就像 object，延迟检查是两者的主要区别。一旦 dynamic 类型上的操作被调用，绑定的结果会被缓存下来以帮助改善下次调用该操作的性能。如果查看动态方法的 IL，将看到 sampleClass 局部变量实际编译成为了 object 类型。

```
.locals init ([0] class [mscorlib]System.Reflection.Assembly asm,
         [1] class [mscorlib]System.Type reflClassType,
         [2] bool V_2,
         [3] object sampleClass)
```

如果我们试图在 SampleClass 上使用 object 代替 dynamic 进行同样的操作：

```
object objSampleClass = Activator.CreateInstance(reflClassType);
Console.WriteLine($"LastMessage: {objSampleClass.LastMessage}");
Console.WriteLine("Calling TestMethod1");
objSampleClass.TestMethod1("Running TestMethod1");
Console.WriteLine($"LastMessage: {objSampleClass.LastMessage}");
Console.WriteLine("Calling TestMethod2");
objSampleClass.TestMethod2("Running TestMethod2", 27);
Console.WriteLine($"LastMessage: {objSampleClass.LastMessage}");
```

会得到以下编译器错误。

```
Error CS1061  'object' does not contain a definition for 'LastMessage' and no
extension method 'LastMessage' accepting a first argument of type 'object' could
be found(are you missing a using directive or an assembly reference ?)
06_ReflectionAndDynamicProgramming.cs  482

Error CS1061  'object' does not contain a definition for 'TestMethod1' and no
extension method 'TestMethod1' accepting a first argument of type 'object' could
be found(are you missing a using directive or an assembly reference ?)
06_ReflectionAndDynamicProgramming.cs  484

Error CS1061  'object' does not contain a definition for 'LastMessage' and no
extension method 'LastMessage' accepting a first argument of type 'object' could
be found(are you missing a using directive or an assembly reference ?)
06_ReflectionAndDynamicProgramming.cs  485

Error CS1061  'object' does not contain a definition for 'TestMethod2' and no
extension method 'TestMethod2' accepting a first argument of type 'object' could
be found(are you missing a using directive or an assembly reference ?)
06_ReflectionAndDynamicProgramming.cs  487

Error CS1061  'object' does not contain a definition for 'LastMessage' and no
extension method 'LastMessage' accepting a first argument of type 'object' could
be found(are you missing a using directive or an assembly reference ?)
06_ReflectionAndDynamicProgramming.cs  488
```

6.7.4　参考

MSDN 文档中的“dynamic”主题。

6.8 动态构建对象

6.8.1 问题

你想要在运行时能够动态构建一个对象并处理它。

6.8.2 解决方案

使用 ExpandoObject 创建一个对象，你可以将属性、方法和事件添加到此对象，并且能够在用户界面中将数据绑定到此对象。

我们可以使用 ExpandoObject 来创建一个初始对象，以持有某人的 Name 和当前 Country，代码如下所示。

```
dynamic expando = new ExpandoObject();
expando.Name = "Brian";
expando.Country = "USA";
```

一旦直接添加了属性，我们还能够通过使用已提供给你的 AddProperty 方法，以更动态的方式向此对象添加属性。有一个例子阐明了你可能这样做的原因，这个例子就是将属性从其他数据源添加到此对象。下列代码将会添加 Langauge 属性。

```
// 将属性动态添加到expando
AddProperty(expando, "Language", "English");
```

AddProperty 方法利用 ExpandoObject 对 IDictionary<string, object> 的支持，使我们能够使用运行时确定的值来添加属性，代码如下所示。

```
public static void AddProperty(ExpandoObject expando, string propertyName,
    object propertyValue)
{
    // ExpandoObject支持IDictionary,所以我们能够如下所示地扩展它
    var expandoDict = expando as IDictionary<string, object>;
    if (expandoDict.ContainsKey(propertyName))
        expandoDict[propertyName] = propertyValue;
    else
        expandoDict.Add(propertyName, propertyValue);
}
```

通过使用代表一个方法调用的 Fun<> 泛型类型，还可以向 ExpandoObject 添加方法。在下面的示例中，我们会将一个验证方法添加到 expando 对象。

```
// 将方法添加到expando
expando.IsValid = (Func<bool>)(() =>
{
    // 检查是否提供了名称
    if(string.IsNullOrWhiteSpace(expando.Name))
        return false;
    return true;
});
```

```
if(!expando.IsValid())
{
    // 不允许继续
}
```

现在，我们还可以使用 Action<> 泛型类型定义事件并将事件添加到 ExpandoObject。我们将添加两个事件，LanguageChanged 和 CountryChanged。在定义 eventHandler 变量来保存 Action<object, EventArgs> 之后，我们将添加 LanguageChanged，还将直接添加 CountryChanged 作为内联匿名方法。CountryChanged 监测 Country，在其改变时以对应 Country 的正确 Language 调用 LanguageChanged 事件。（注意，LanguageChanged 也是一个匿名方法，但有时添加一个对应的变量能够使代码更清晰。）

```
// 同样可以将事件处理器添加到expando对象
var eventHandler =
    new Action<object, EventArgs>((sender, eventArgs) =>
    {
        dynamic exp = sender as ExpandoObject;
        var langArgs = eventArgs as LanguageChangedEventArgs;
        Console.WriteLine($"Setting Language to : {langArgs?.Language}");
        exp.Language = langArgs?.Language;
    });

// 添加一个LanguageChanged事件和预定义的事件处理器
AddEvent(expando, "LanguageChanged", eventHandler);

// 添加一个CountryChanged事件和一个内联事件处理器
AddEvent(expando, "CountryChanged",
    new Action<object, EventArgs>((sender, eventArgs) =>
{
    dynamic exp = sender as ExpandoObject;
    var ctryArgs = eventArgs as CountryChangedEventArgs;
    string newLanguage = string.Empty;
    switch (ctryArgs?.Country)
    {
        case "France":
            newLanguage = "French";
            break;
        case "China":
            newLanguage = "Mandarin";
            break;
        case "Spain":
            newLanguage = "Spanish";
            break;
    }
    Console.WriteLine($"Country changed to {ctryArgs?.Country}, " +
        $"changing Language to {newLanguage}");
    exp?.LanguageChanged(sender,
        new LanguageChangedEventArgs() { Language = newLanguage });
}));
```

下列代码提供了 AddEvent 方法以封装将事件添加到 ExpandoObject 的细节。这再一次利用了 ExpandoObject 对 IDictionary<string, object> 的支持。

```
public static void AddEvent(ExpandoObject expando, string eventName,
Action<object, EventArgs> handler)
{
    var expandoDict = expando as IDictionary<string, object>;
    if (expandoDict.ContainsKey(eventName))
        expandoDict[eventName] = handler;
    else
        expandoDict.Add(eventName, handler);
}
```

最后，ExpandoObject 支持 INotifyPropertyChanged，这是 .NET 中将数据绑定到属性的基础。我们挂接事件处理程序，当 Country 属性更改时触发 CountryChanged 事件。

```
 ((INotifyPropertyChanged)expando).PropertyChanged +=
    new PropertyChangedEventHandler((sender, ea) =>
{
    dynamic exp = sender as dynamic;
    var pcea = ea as PropertyChangedEventArgs;
    if(pcea?.PropertyName == "Country")
        exp.CountryChanged(exp, new CountryChangedEventArgs()
            { Country = exp.Country });
});
```

现在，我们已经完成了对象的构建，可以像下列代码一样调用它来模拟我们周游世界的朋友。

```
Console.WriteLine($"expando contains: {expando.Name}, {expando.Country}, " +
    $"{expando.Language}");
Console.WriteLine();

Console.WriteLine("Changing Country to France...");
expando.Country = "France";
Console.WriteLine($"expando contains: {expando.Name}, {expando.Country},  " +
    $"{expando.Language}");
Console.WriteLine();

Console.WriteLine("Changing Country to China...");
expando.Country = "China";
Console.WriteLine($"expando contains: {expando.Name}, {expando.Country},  " +
    $"{expando.Language}");
Console.WriteLine();

Console.WriteLine("Changing Country to Spain...");
expando.Country = "Spain";
Console.WriteLine($"expando contains: {expando.Name}, {expando.Country},  " +
    $"{expando.Language}");
Console.WriteLine();
```

此示例的输出如下所示。

```
expando contains: Brian, USA, English

Changing Country to France...
Country changed to France, changing Language to French
Setting Language to: French
```

```
expando contains: Brian, France, French

Changing Country to China...
Country changed to China, changing Language to Mandarin
Setting Language to: Mandarin
expando contains: Brian, China, Mandarin

Changing Country to Spain...
Country changed to Spain, changing Language to Spanish
Setting Language to: Spanish
expando contains: Brian, Spain, Spanish
```

6.8.3 讨论

ExpandoObject 允许编写比使用 GetProperty("Field") 的典型反射代码更具可读性的代码。当处理 XML 或 JSON 时，ExpandoObject 可用于快速设置一个类型来编程，而非总是必须创建数据传输对象。对于任何使用 WPF、MVC 或其他 .NET 中的编定框架的开发人员来说，ExpandoObject 通过 INotifyPropertyChanged 对数据绑定的支持是最大的亮点，它允许你像使用其他静态类型一样使用这些对象，而且是动态的。

因为 ExpandoObject 将委托作为其成员，所以你可以将方法和事件附加到这些动态的类型，同时代码看起来像在处理一个静态类型。

```
public static void AddEvent(ExpandoObject expando, string eventName,
Action<object, EventArgs> handler)
{
    var expandoDict = expando as IDictionary<string, object>;
    if (expandoDict.ContainsKey(eventName))
        expandoDict[eventName] = handler;
    else
        expandoDict.Add(eventName, handler);
}
```

你可能想知道为什么我们没有将 AddProperty 和 AddEvent 编写为扩展方法。它们都可以挂接在 ExpandoObject 上并使语法更简洁，对吧？不幸的是，不能。扩展方法的工作方式是编译器搜索所有可能匹配扩展类的类。这意味着，DLR 必须在运行时同样知道所有这些信息（因为 ExpandoObject 由 DLR 处理），而目前并不是类和方法所有的信息都被编码到了调用站点。

下面列出了 LanguageChanged 和 CountryChanged 事件的事件参数类。

```
public class LanguageChangedEventArgs : EventArgs
{
    public string Language { get; set; }
}

public class CountryChangedEventArgs : EventArgs
{
    public string Country { get; set; }
}
```

6.8.4 参考

MSDN 文档中的“ExpandoObject 类”“Func<> 委托”“Action<> 委托”和“INotifyPropertyChanged 接口”主题。

6.9 使对象可扩展

6.9.1 问题

你想要有一个基类可用于在运行时扩展对象，以便可以从此基类派生出多个模型以避免重复的代码。

6.9.2 解决方案

使用从 DynamicObject 派生的 DynamicBase<T> 类来创建一个新类或者封装一个现有类。

```
public class DynamicBase<T> : DynamicObject
    where T : new()
{
    private T _containedObject = default(T);

    [JsonExtensionData] //JSON.NET 5.0或更高版本
    private Dictionary<string, object> _dynamicMembers =
        new Dictionary<string, object>();

    private List<PropertyInfo> _propertyInfos =
        new List<PropertyInfo>(typeof(T).GetProperties());

    public DynamicBase()
    {
    }
    public DynamicBase(T containedObject)
    {
        _containedObject = containedObject;
    }

    public override bool TryInvokeMember(InvokeMemberBinder binder,
        object[] args, out object result)
    {
        if (_dynamicMembers.ContainsKey(binder.Name)
        && _dynamicMembers[binder.Name] is Delegate)
        {
            result = (_dynamicMembers[binder.Name] as Delegate).DynamicInvoke(
                args);
            return true;
        }

        return base.TryInvokeMember(binder, args, out result);
    }

    public override IEnumerable<string> GetDynamicMemberNames() =>
```

```
        _dynamicMembers.Keys;

    public override bool TryGetMember(GetMemberBinder binder, out object result)
    {
        result = null;
        var propertyInfo = _propertyInfos.Where(pi =>
            pi.Name == binder.Name).FirstOrDefault();
        // 确认此成员并不是对象上已有的属性
        if (propertyInfo == null)
        {
            // 在额外数据项集合中查找它
            if (_dynamicMembers.Keys.Contains(binder.Name))
            {
                // 返回动态数据项
                result = _dynamicMembers[binder.Name];
                return true;
            }
        }
        else
        {
            // 从包含的对象中获得它
            if (_containedObject != null)
            {
                result = propertyInfo.GetValue(_containedObject);
                return true;
            }
        }
        return base.TryGetMember(binder, out result);
    }

    public override bool TrySetMember(SetMemberBinder binder, object value)
    {
        var propertyInfo = _propertyInfos.Where(pi =>
            pi.Name == binder.Name).FirstOrDefault();
        // 确认此成员并不是对象上已有的属性
        if (propertyInfo == null)
        {
            // 在额外数据项集合中查找它
            if (_dynamicMembers.Keys.Contains(binder.Name))
            {
                // 设置动态数据项
                _dynamicMembers[binder.Name] = value;
                return true;
            }
            else
            {
                _dynamicMembers.Add(binder.Name, value);
                return true;
            }
        }
        else
        {
            // 将其置于包含的对象中
            if (_containedObject != null)
            {
```

```
                propertyInfo.SetValue(_containedObject, value);
                return true;
            }
        }
        return base.TrySetMember(binder, value);
    }

    public override string ToString()
    {
        StringBuilder builder = new StringBuilder();
        foreach (var propInfo in _propertyInfos)
        {
            if(_containedObject != null)
                builder.AppendFormat("{0}:{1}{2}", propInfo.Name,
                   propInfo.GetValue(_containedObject), Environment.NewLine);
            else
                builder.AppendFormat("{0}:{1}{2}", propInfo.Name,
                   propInfo.GetValue(this), Environment.NewLine);
        }
        foreach (var addlItem in _dynamicMembers)
        {
            // 排除从描述中添加的方法
            Type itemType = addlItem.Value.GetType();
            Type genericType =
                itemType.IsGenericType ?
                    itemType.GetGenericTypeDefinition() : null;
            if (genericType != null)
            {
                if (genericType != typeof(Func<>) &&
                    genericType != typeof(Action<>))
                    builder.AppendFormat("{0}:{1}{2}", addlItem.Key,
                        addlItem.Value, Environment.NewLine);
            }
            else
                builder.AppendFormat("{0}:{1}{2}", addlItem.Key, addlItem.Value,
                    Environment.NewLine);
        }
        return builder.ToString();
    }
}
```

要理解如何使用 DynamicBase<T>，设想我们有一个接收包含运动员信息的 JSON 序列化负载的 Web 服务。我们目前已经定义了包括 Name 和 Sport 属性的 DynamicAthlete 类。

```
public class DynamicAthlete : DynamicBase<DynamicAthlete>
{
    public string Name { get; set; }
    public string Sport { get; set; }
}
```

在发送给我们的负载中，调用者已经开始发送关于运动员的 Position 的额外信息。在遗留系统集成发生变化而所有系统不能在同一时间都更新时，就会发生这种情况。对于我们的接收系统，我们不想丢失某些系统正在发送的新数据。我们使用 dynamic 和 JSON.NET 序列化器来模拟构造 JSON 负载，JSON.NET 可通过 NuGet 获得。（谢谢你，James Newton-

King，这太棒了！）

```
// 创建运动员的一组信息
// 注意接收信息的服务并没有在Athlete对象上具有Position属性
dynamic initialAthletes = new[]
{
    new
    {
        Name = "Tom Brady",
        Sport = "Football",
        Position = "Quarterback"
    },
    new
    {
        Name = "Derek Jeter",
        Sport = "Baseball",
        Position = "Shortstop"
    },
    new
    {
        Name = "Michael Jordan",
        Sport = "Basketball",
        Position = "Small Forward"
    },
    new
    {
        Name = "Lionel Messi",
        Sport = "Soccer",
        Position = "Forward"
    }
};

// 使JSON序列化以发送到关于运动员的一个Web服务
string serializedAthletes = JsonNetSerialize(initialAthletes);
```

假定运动员信息的 JSON 负载进入你的服务并且被反序列化（再次感谢 JSON.NET），然后我们将其反序列化为一个 DynamicAthlete 数组，代码如下所示。

```
// 反序列化发送的JSON
var athletes = JsonNetDeserialize<DynamicAthlete[]>(serializedAthletes);
```

现在，每一个开发过任何类型 Web 服务的开发人员（或者任何序列化的开发，就此而论）都知道，如果你没有一个地方放置反序列化的信息，将会丢失信息或者导致错误。那么既然 DynamicAthlete 上没有声明 Position，传入的 Position 属性值会发生什么呢？如果回头看一下 DynamicBase<T> 的声明（DynamicAthlete 从其派生），将会看到一个内部的私有 Dictionary<string,object>，它被标记有 JsonExtensionData 特性。此特性告诉序列化器在何处存放在派生对象中没有对应位置的属性值。这太厉害了！我们的 Position 值存储在这个内部的字典中，这非常棒，但我们如何访问它呢？

```
[JsonExtensionData] //JSON.NET 5.0及以上
private Dictionary<string, object> _dynamicMembers =
    new Dictionary<string, object>();
```

既然 DynamicAthlete 派生自 DynamicBase<T>，而 DynamicBase<T> 又是从 DynamicObject 派生的，我们可以将接收到的第一位运动员赋值到动态变量 da。一旦存到一个动态变量中，我们就可以访问 Position，就好像它是在 DynamicAthlete 中定义的属性之一。

```
dynamic da = athletes[0];
Console.WriteLine($"Position of first athlete: {da.Position}");
```

因此，可以保留发送给我们的属性值，即使当部署服务时我们并不直接了解它们，这是一个很好的可靠性特征。我们还可以向每个 DynamicAthlete 添加一个新方法，以获取 Name 的大写形式，同时输出收到的内容。

```
// 检视athletes并注意到不仅获得Position
// 信息,而且还可以添加一个操作作用于实体
// 并作为动态实体的一部分调用该操作
foreach(var athlete in athletes)
{
    dynamic dynamicAthlete = (dynamic)athlete;
    dynamicAthlete.GetUppercaseName =
        (Func<string>)(() =>
        {
            return ((string)dynamicAthlete.Name).ToUpper();
        });
    Console.WriteLine($"Athlete:");
    Console.WriteLine(athlete);
    Console.WriteLine($"Uppercase Name: {dynamicAthlete.GetUppercaseName()}");
    Console.WriteLine();
    Console.WriteLine();
}
```

GetUppercaseName 被添加到对象，然后被调用以返回 Name 的大写版本。相应的输出如下所示。

```
Athlete:
Name:Tom Brady
Sport:Football
Position:Quarterback

Uppercase Name: TOM BRADY


Athlete:
Name:Derek Jeter
Sport:Baseball
Position:Shortstop

Uppercase Name: DEREK JETER


Athlete:
Name:Michael Jordan
Sport:Basketball
Position:Small Forward

Uppercase Name: MICHAEL JORDAN
```

```
Athlete:
Name:Lionel Messi
Sport:Soccer
Position:Forward

Uppercase Name: LIONEL MESSI
```

我们已经定义了对象的情况怎么样呢？如何能够获得这一扩展的优势呢？我们看一下 StaticAthlete 类作为一个例子的情形。

```
public class StaticAthlete
{
    public string Name { get; set; }
    public string Sport { get; set; }
}
```

StaticAthlete 看起来与 DynamicAthlete 差不多，但它不从任何类派生。

如果创建了一个 StaticAthlete 实例，我们仍然可以使用 DynamicBase<T> 将其包装并获取相同的扩展行为，如同我们将 DynamicAthlete 从 DynamicBase<T> 派生时所做的一样。

```
//包装一个已有的athlete
StaticAthlete staticAthlete = new StaticAthlete()
{
    Sport = "Hockey"
};

dynamic extendedAthlete = new DynamicBase<StaticAthlete>(staticAthlete);
extendedAthlete.Name = "Bobby Orr";
extendedAthlete.Position = "Defenseman";
extendedAthlete.GetUppercaseName =
        (Func<string>)(() =>
        {
            return ((string)extendedAthlete.Name).ToUpper();
        });
Console.WriteLine($"Static Athlete (extended):");
Console.WriteLine(extendedAthlete);
Console.WriteLine($"Uppercase Name: {extendedAthlete.GetUppercaseName()}");
Console.WriteLine();
Console.WriteLine();
```

你可以看到 StaticAthlete 的输出与 DynamicAthlete 的输出是完全相同的。

```
Static Athlete (extended):
Name:Bobby Orr
Sport:Hockey
Position:Defenseman

Uppercase Name: BOBBY ORR
```

6.9.3 讨论

DynamicObject 作为一个基类来帮助你向类中添加动态行为。与 ExpandoObject 不同，它不

能被实例化，但可以从它派生。使用 DynamicObject 可以重写许多不同类型的操作，如属性或方法访问以及任意二元、一元或类型转换操作，使得你能够灵活地确定类在运行时是如何反应的。

我们在 DynamicBase<T> 中通过重写以下 DynamicObject 的方法实现了其中一些操作。

- TryInvokeMember
- GetDynamicMemberNames
- TryGetMember
- TrySetMember

TryInvokeMember 使得我们能够确定当在对象上调用成员时，应该发生什么事。在 DynamicBase<T> 中用它来查看内部集合，如果有一个匹配的数据项，就将其作为一个委托来动态调用它。

```
    public override bool TryInvokeMember(InvokeMemberBinder binder,
        object[] args,
out object result)
    {
        if (_dynamicMembers.ContainsKey(binder.Name) &&
            _dynamicMembers[binder.Name] is Delegate)
        {
            result = (_dynamicMembers[binder.Name] as Delegate).DynamicInvoke(
                args);
            return true;
        }

        return base.TryInvokeMember(binder, args, out result);
    }
```

GetDynamicMemberNames 获取动态添加的所有成员的集合，代码如下所示。

```
public override IEnumerable<string> GetDynamicMemberNames()
{
    return _dynamicMembers.Keys;
}
```

重写了 TryGetMember 以允许调用方获取动态添加项目的属性值。如果类的主要属性信息中找不到它，就在包含动态成员的内部字典中查找，并从那里返回它，代码如下所示。

```
public override bool TryGetMember(GetMemberBinder binder, out object result)
{
    result = null;
    var propertyInfo = _propertyInfos.Where(pi =>
        pi.Name == binder.Name).FirstOrDefault();
    // 确认此成员并不是对象上的属性
    if (propertyInfo == null)
    {
        // 在额外的数据项集合中查找它
        if (_dynamicMembers.Keys.Contains(binder.Name))
        {
            // 返回动态数据项
            result = _dynamicMembers[binder.Name];
```

```
                return true;
            }
        }
        else
        {
            // 从包含的对象中获得属性
            if (_containedObject != null)
            {
                result = propertyInfo.GetValue(_containedObject);
                return true;
            }
        }
        return base.TryGetMember(binder, out result);
    }
```

TrySetMember 的重写处理了属性值的设置操作。同上次一样，先查看类型化的对象，然后到动态字典中查找在哪里存储值，代码如下所示。

```
    public override bool TrySetMember(SetMemberBinder binder, object value)
    {
        var propertyInfo = _propertyInfos.Where(pi =>
            pi.Name == binder.Name).FirstOrDefault();
        // 确认此成员并不是对象上的属性
        if (propertyInfo == null)
        {
            // 在额外的数据项集合中查找它
            if (_dynamicMembers.Keys.Contains(binder.Name))
            {
                // 设置动态数据项
                _dynamicMembers[binder.Name] = value;
                return true;
            }
            else
            {
                _dynamicMembers.Add(binder.Name, value);
                return true;
            }
        }
        else
        {
            // 设置到包含的对象中
            if (_containedObject != null)
            {
                propertyInfo.SetValue(_containedObject, value);
                return true;
            }
        }
        return base.TrySetMember(binder, value);
    }
```

我们也重写了 ToString，以便能获得要在字符串中描述的类的所有属性（静态和动态），代码如下所示。

```
    public override string ToString()
    {
```

```
        StringBuilder builder = new StringBuilder();
        foreach (var propInfo in _propertyInfos)
        {
            if(_containedObject != null)
                builder.AppendFormat("{0}:{1}{2}", propInfo.Name,
                    propInfo.GetValue(_containedObject), Environment.NewLine);
            else
                builder.AppendFormat("{0}:{1}{2}", propInfo.Name,
                    propInfo.GetValue(this), Environment.NewLine);
        }
        foreach (var addlItem in _dynamicMembers)
        {
            // 排除描述中添加进来的方法
            Type itemType = addlItem.Value.GetType();
            Type genericType =
                itemType.IsGenericType ? itemType.GetGenericTypeDefinition() : null;
            if (genericType != null)
            {
                if (genericType != typeof(Func<>) &&
                    genericType != typeof(Action<>))
                    builder.AppendFormat("{0}:{1}{2}", addlItem.Key, addlItem.Value,
                        Environment.NewLine);
            }
            else
                builder.AppendFormat("{0}:{1}{2}", addlItem.Key, addlItem.Value,
                    Environment.NewLine);
        }
        return builder.ToString();
    }
```

我们做了一点筛选，以处理动态方法和事件被添加进来以及成员或方法位于包含的对象上的情况。这使得我们能够得到所有属性的描述，如下所示。

```
Name:Bobby Orr
Sport:Hockey
Position:Defenseman
```

可以看到，`DynamicObject` 给予了你尽你所想地去扩展对象所需的所有能力。

6.9.4　参考

MSDN 文档中的“`DynamicObject` 类”主题。

第 7 章

正则表达式

7.0 简介

.NET Framework 类库（FCL）包括 `System.Text.RegularExpressions` 命名空间，它专用于创建、执行用于字符串的正则表达式，并获得其执行结果。

正则表达式采用模式形式，模式可以与字符串内的 0 个、1 个或多个字符匹配。最简单的一些模式是非常易学的，例如 `.*`（匹配除换行符外的任何内容）和 `[A-Za-z]`（匹配任何字符），但是更高级的模式学习起来会比较困难，要正确地实现它们甚至会更困难。学习和理解正则表达式可能需要花费相当多的时间和精力，但是非常值得。

> Michael Fitzgeraldr 所著的《学习正则表达式》与 Jan Goyvaerts 和 Steven Levithan 合著的《正则表达式经典实例》可以帮助你学习和扩展对正则表达式的理解。这两本英文原书都是由 O'Reilly 出版的。

正则表达式模式可以具有简单的形式（比如一个单词或字符）或者复杂得多的模式。更复杂的模式可以识别和匹配很多内容，例如，日期中的年份、ASP 页面中的所有 `<SCRIPT>` 标签、句子中的短语（根据每次使用区分它们）。.NET 正则表达式提供了一种非常灵活、强大的方式来执行许多任务，例如，识别文本、替换字符串内的文本、基于一个或多个复杂的分隔符把文本分割成单独的部分。

尽管正则表达式模式很复杂，但是 FCL 中的正则表达式类很易于在应用程序中使用。执行一个正则表达式包括以下步骤。

(1) 创建 `Regex` 对象的一个实例，它包含正则表达式模式以及用于执行该模式的任何选项。
(2) 如果只需要第一个找到的匹配，可以通过调用 `Match` 实例方法，获取指向 `Match` 对象的

实例引用。如果想要的不仅仅是第一个找到的匹配，可以通过调用 Mathches 实例方法，获取指向 MatchesCollection 对象的实例引用。不过，如果只想知道输入字符串是否是一个匹配，并且不需要关于匹配的额外细节，则可使用 Regex.IsMatch 方法。

(3) 如果调用 Matches 方法获取一个 MatchCollection 对象，可使用 foreach 循环遍历 MatchCollection。每次迭代都允许访问正则表达式产生的每一个 Match 对象。

7.1 从MatchCollection中提取组

7.1.1 问题

你有一个正则表达式，其中包含一个或多个命名组（也被称为命名捕获组），如下所示。

```
\\\\(?<TheServer>\w*)\\(?<TheService>\w*)\\
```

其中命名组 TheServer 将匹配 UNC 字符串内的任何服务器名称，TheService 将匹配 UNC 字符串内的任何服务名称。

该模式与 UNCW 格式不匹配。

需要把这个正则表达式返回的组存储在一个通过键访问的集合（比如 Dictionary<string, Group>）中，其中的键是组名称。

7.1.2 解决方案

例 7-1 所示的 ExtractGroupings 方法获得以匹配的组名称为键的一组 Group 对象。

例 7-1：ExtractGroupings 方法

```
using System;
using System.Collections;
using System.Collections.Generics;
using System.Text.RegularExpressions;

public static List<Dictionary<string, Group>> ExtractGroupings(string source
                                                              string matchPattern,
                                                              bool wantInitialMatch)
{
    List<Dictionary<string, Group>> keyedMatches =
        new List<Dictionary<string, Group>>();
    int startingElement = 1;
    if (wantInitialMatch)
    {
        startingElement = 0;
    }

    Regex RE = new Regex(matchPattern, RegexOptions.Multiline);
    MatchCollection theMatches = RE.Matches(source);
```

```
        foreach(Match m in theMatches)
        {
            Dictionary<string, Group> groupings = new Dictionary<string, Group>();

            for (int counter = startingElement; counter < m.Groups.Count; counter++)
            {
                // 如果只是直接返回MatchCollection，
                // GroupNameFromNumber方法将不可用
                groupings.Add(RE.GroupNameFromNumber(counter), m.Groups[counter]);
            }
            keyedMatches.Add(groupings);
        }
        return (keyedMatches);
    }
```

可以用下列方式使用 ExtractGroupings 方法，提取命名组并按名称组织它们。

```
    public static void TestExtractGroupings()
    {
        string source = @"Path = ""\\MyServer\MyService\MyPath;
                              \\MyServer2\MyService2\MyPath2\""";
        string matchPattern = @"\\\\(?<TheServer>\w*)\\(?<TheService>\w*)\\";

        foreach (Dictionary<string, Group> grouping in
                 ExtractGroupings(source, matchPattern, true))
        {
            foreach (KeyValuePair<string, Group> kvp in grouping)
                Console.WriteLine($"Key/Value = {kvp.Key} / {kvp.Value}");
            Console.WriteLine("");
        }
    }
```

这个测试方法将创建一个 source 字符串并在 matchPattern 变量中创建一个正则表达式模式。下面突出显示了这个正则表达式中的两个分组。

```
    string matchPattern = @"\\\\(?<TheServer>\w*)\\(?<TheService>\w*)\\";
```

这两个组的名称是 TheServer 和 TheService。可以通过这些组名称访问与其中任何一个分组匹配的文本。

将 source 和 matchPattern 变量以及一个布尔值传入 ExtractGroupings 方法中，稍后将讨论这个布尔值。该方法返回一个包含 Dictionary<string, Group> 对象的 List<T>。这些 Dictionary<string, Group> 对象包含正则表达式中每个命名组的匹配，并以它们的组名称为键。

测试方法 TestExtractGroupings 返回如下内容。

```
    Key / Value = 0 / \\MyServer\MyService\
    Key / Value = TheService / MyService
    Key / Value = TheServer / MyServer

    Key / Value = 0 / \\MyServer2\MyService2\
    Key / Value = TheService / MyService2
    Key / Value = TheServer / MyServer2
```

如果把 ExtractGroupings 方法的最后一个参数修改为 false，将产生下面的输出。

```
Key / Value = TheService / MyService
Key / Value = TheServer / MyServer

Key / Value = TheService / MyService2
Key / Value = TheServer / MyServer2
```

这两个输出之间的唯一区别是，当把 ExtractGroupings 的最后一个参数改为 false 时不会显示第一个分组。第一个分组总是正则表达式的完全匹配。

7.1.3 讨论

可以用两种方式之一定义正则表达内的组。第一种方式是，用圆括号括住你希望定义为分组的子模式。这种分组类型有时被称为*未命名的*（unnamed）。之后可以轻松地从运行正则表达式返回的每个 Match 对象中的最终文本中提取这些分组。可以像下面这样修改用于本范例的正则表达式，以使用简单的未命名组。

```
string matchPattern = @"\\\\(\w*)\\(\w*)\\";
```

在运行正则表达式后，可以使用以 1 开始的整数值访问这些组。

在正则表达式内定义组的第二种方式是使用一个或多个命名组。定义命名组的方式如下：使用以下语法，用圆括号括住你希望定义为分组的子模式，并给每个分组添加名称。

```
(?<Name>\w*)
```

此语法的 *Name* 部分是为这个组指定的名称。在执行这个正则表达式之后，可以通过名称 *Name* 访问该组。

为了访问每个组，首先必须使用一个循环来迭代 MatchCollection 中的每个 Match 对象。对于每个 Match 对象，可以使用以下未命名的语法访问 GroupCollection 的索引器。

```
string group1 = m.Groups[1].Value;
string group2 = m.Groups[2].Value;
```

也可以使用下面命名组的语法访问它，其中 m 是 Match 对象。

```
string group1 = m.Groups["Group1_Name"].Value;
string group2 = m.Groups["Group2_Name"].Value;
```

如果使用 Match 方法返回单个 Match 对象而不是返回 MatchCollection，则可使用下面的语法访问每个组：

```
// 未命名组的语法
string group1 = theMatch.Groups[1].Value;
string group2 = theMatch.Groups[2].Value;

// 命名组的语法
string group1 = theMatch.Groups["Group1_Name"].Value;
string group2 = theMatch.Groups["Group2_Name"].Value;
```

其中 theMatch 是 Match 方法返回的 Match 对象。

7.1.4 参考

MSDN 文档中的“.NET Framework 正则表达式”和“Dictionary 类”主题。

7.2 验证正则表达式的语法

7.2.1 问题

你通过代码或者基于用户输入动态构建了一个正则表达式。在实际使用它之前，你需要测试这个正则表达式语法的有效性。

7.2.2 解决方案

使用例 7-2 中所示的 VerifyRegEx 方法，测试正则表达式语法的有效性。

例 7-2：VerifyRegEx 方法

```
using System;
using System.Text.RegularExpressions;

public static bool VerifyRegEx(string testPattern)
{
    bool isValid = true;
    if ((testPattern?.Length ?? 0) > 0)
    {
        try
        {
            Regex.Match("", testPattern);
        }
        catch (ArgumentException)
        {
            // 坏模式:语法错误
            isValid = false;
        }
    }
    else
    {
        // 坏模式:模式为null或空字符串
        isValid = false;
    }

    return (isValid);
}
```

要使用这个方法，可以把希望验证的正则表达式传递给它，代码如下所示。

```
public static void TestUserInputRegEx(string regEx)
{
    if (VerifyRegEx(regEx))
        Console.WriteLine("This is a valid regular expression.");
    else
        Console.WriteLine("This is not a valid regular expression.");
}
```

7.2.3　讨论

VerifyRegEx 方法调用静态方法 Regex.Match，这一静态方法用于直接对字符串运行正则表达式模式。静态方法 RegEx.Match 返回一个 Match 对象。通过使用该静态方法对字符串（在本示例中是一个空字符串）运行正则表达式，可以通过监视引发的异常来确定正则表达式是否有效。如果正则表达式的语法不正确，Regex.Match 方法将会引发一个 ArgumentException。这个异常的 Message 属性包含正则表达式运行失败的原因，ParamName 属性则包含传递给 Match 方法的正则表达式。这两个属性都是只读的。

在利用静态方法 Match 测试正则表达式之前，VerifyRegEx 方法先测试正则表达式是否为 null 或者为空。当把值为 null 的正则表达式字符串传入 Match 方法时，它会引发一个 ArgumentNullException。另一方面，如果把空的正则表达式传入 Match 方法，将不会引发异常（只要同时把一个有效的字符串传递给了 Match 方法的第一个参数）。

虽然这个范例可以验证正则表达式语法是否正确，但它不会查找写得很糟糕的表达式。糟糕正则表达式的一个常见情况是，表达式依赖回溯功能。回溯会导致正则表达式需要以指数倍增的时间才能完成，使得看起来好像执行正则表达式的代码不动了。

> 关于正则表达式回溯的详细说明，可参阅 MSDN 文档中的“正则表达式中的回溯”主题，该主题位于“.NET Framework 正则表达式”父主题之下。

在正则表达式使用回溯的情况下，建议你使用超时值来限制一个正则表达式必须完成的时间。使用以下正则表达式构造函数。

```
Regex (String, RegexOptions, TimeSpan)
```

其中 TimeSpan 是允许正则表达式执行的时间长度。

```
Regex regex = new RegEx(bkTrkPattern, RegexOptions.None,
                        TimeSpan.FromMilliseconds(1000));
```

然后，可以在一个 try-catch 语句块中执行正则表达式，使用 RegexMatchTimeoutException 来捕获一个需要非常长时间执行的糟糕正则表达式。

7.3　增强基本的字符串替换函数

7.3.1　问题

你需要用一个新字符串替换目录字符串内的字符模式。不过，在此例中，每一次替换操作都有一组必须满足的独特条件以允许执行替换。

7.3.2　解决方案

使用例 7-3 中所示的重载 Replace 实例方法，它接受一个 MatchEvaluator 委托以及其他参

数。MatchEvaluator 委托是一个回调方法，它重写了 Replace 方法的默认行为。

例 7-3：接受一个 MatchEvaluator 委托的重载 Replace 方法

```
using System;
using System.Text.RegularExpressions;

public static string MatchHandler(Match theMatch)
{
    // 处理所有ControlID_记录
    if (theMatch.Value.StartsWith("ControlID_", StringComparison.Ordinal))
    {
        long controlValue = 0;

        // 获得Top属性的数值
        Match topAttributeMatch = Regex.Match(theMatch.Value, "Top=([-]*\\d*)");
        if (topAttributeMatch.Success)
        {
            if (topAttributeMatch.Groups[1].Value.Trim().Equals(""))
            {
                // 如果为空,设置为0
                return (theMatch.Value.Replace(
                        topAttributeMatch.Groups[0].Value.Trim(),
                        "Top=0"));
            }
            else if (topAttributeMatch.Groups[1].Value.Trim().StartsWith("-"
                                         , StringComparison.Ordinal))
            {
                // 如果只有一个负号(语法错误),设置为0
                return (theMatch.Value.Replace(
                        topAttributeMatch.Groups[0].Value.Trim(), "Top=0"));
            }
            else
            {
                // 获得了一个有效的数字
                // 将匹配的字符串转换为数字
                 controlValue = long.Parse(topAttributeMatch.Groups[1].Value,
                           System.Globalization.NumberStyles.Any);
                // 如果Top属性超出了指定的范围
                // 将其设置为0
                if (controlValue < 0 || controlValue > 5000)
                {
                    return (theMatch.Value.Replace(
                            topAttributeMatch.Groups[0].Value.Trim(),
                            "Top=0"));
                }
            }
        }
    }
    return (theMatch.Value);
}
```

Replace 方法的回调方法如下所示。

```
public static void ComplexReplace(string matchPattern, string source)
{
```

```
        MatchEvaluator replaceCallback = new MatchEvaluator(MatchHandler);
        Regex RE = new Regex(matchPattern, RegexOptions.Multiline);
        string newString = RE.Replace(source, replaceCallback);

        Console.WriteLine($"Replaced String = {newString}");
    }
```

为了将这个回调方法与 Replace 静态方法结合使用，可以修改前面的 ComplexReplace 方法，如下所示。

```
    public void ComplexReplace(string matchPattern, string source)
    {
        MatchEvaluator replaceCallback = new MatchEvaluator(MatchHandler);
        string newString = Regex.Replace(source, matchPattern, replaceCallback);
        Console.WriteLine("Replaced String = " + newString);
    }
```

其中 source 是要对其运行替换操作的原始字符串，matchPattern 是 source 字符串中要匹配的正则表达式模式。

如果通过以下代码调用 ComplexReplace 方法：

```
    public static void TestComplexReplace()
    {
        string matchPattern = "(ControlID_.*)";
        string source = @"WindowID=Main
        ControlID_TextBox1 Top=-100 Left=0 Text=BLANK
        ControlID_Label1 Top=9999990 Left=0 Caption=Enter Name Here
        ControlID_Label2 Top= Left=0 Caption=Enter Name Here";

        ComplexReplace(matchPattern, source);
    }
```

将只会把 ControlID_* 行的 Top 属性从它们的原始值更改为 0。

如果 ControlID_* 行的 Top 属值性小于 0 或大于 5000，这个替换操作的结果将把该属性值更改为 0。包含 Top 属性的任何其他标签都将保持不变。下面三行 source 字符串将从：

```
ControlID_TextBox1 Top=-100 Left=0 Text=BLANK
ControlID_Label1 Top=9999990 Left=0 Caption=Enter Name Here
ControlID_Label2 Top= Left=0 Caption=Enter Name Here";
```

更改为：

```
ControlID_TextBox1 Top=0 Left=0 Text=BLANK
ControlID_Label1 Top=0 Left=0 Caption=Enter Name Here
ControlID_Label2 Top=0 Left=0 Caption=Enter Name Here";
```

7.3.3　讨论

在将 MatchEvaluator 委托作为参数提供给 Regex 类的 Replace 方法时将自动调用它，允许对符合正则表达式模式的每个字符串执行自定义替换。

如果当前 Match 对象操作的 ControlID_* 行的 Top 属性超出了指定的范围，MatchHandler

回调方法内的代码将会返回修改过的新字符串。否则，就会不加修改地返回当前匹配的字符串。这一特性允许重写默认的 Replace 功能，只替换 source 字符串中满足某种条件的那一部分。这个回调方法内的代码将让你理解可以使用这种替换技术完成什么任务。

为了使用这个回调方法，需要采用一种方式从 ComplexReplace 方法内调用它。首先，创建一个 System.Text.RegularExpressions.MatchEvaluator 类型的变量。该变量（replaceCallback）是用于调用 MatchHandler 方法的委托。

```
MatchEvaluator replaceCallback = new MatchEvaluator(MatchHandler);
```

最后，将 MatchEvaluator 委托的引用作为参数传入 Replace 方法调用，代码如下所示。

```
string newString = Regex.Replace(source, matchPattern, replaceCallback);
```

7.3.4 参考

MSDN 文档中的“.NET Framework 正则表达式”主题。

7.4 实现一个更好的分词器

7.4.1 问题

你需要一个分词器（tokenizer），也被称为词法分析器（lexer），它可以基于一组明确定义的字符来分割字符串。

7.4.2 解决方案

使用 Regex 类的 Split 方法，可以创建正则表达式来指示你有兴趣收集的标记或者分隔符的类型。这种技术特别适合方程式，因为方程式的标记是明确定义的，如下面的代码所示。

```
using System;
using System.Text.RegularExpressions;

public static string[] Tokenize(string equation)
{
    Regex re = new Regex(@"([\+\-\*\(\)\^\\])");
    return (re.Split(equation));
}
```

上述代码将依据 Regex 构造函数中指定的正则表达式分割一个字符串。换句说，将基于分隔符 +、-、*、(、)、^ 和 \ 来分割传入 Tokenize 方法中的字符串。下面的方法将调用 Tokenize 方法来标记方程式 (y - 3)*(3111*x^21 + x + 320)。

```
public static void TestTokenize()
{
    foreach(string token in Tokenize("(y - 3)*(3111*x^21 + x + 320)"))
        Console.WriteLine("String token = " + token.Trim());
}
```

这将显示以下输出。

```
string token =
String token = (
String token = y
String token = -
String token = 3
String token = )
String token = *
String token = (
String token = 3111
String token = *
String token = x
String token = ^
String token = 21
String token = +
String token = x
String token = +
String token = 320
String token = )
String token =
```

注意每个单独的运算符、圆括号和数字都被分解成它自己单独的标记。

7.4.3　讨论

在现实的项目中，并不是总能够控制代码的输入集。通过利用正则表达式，可以采取原始的分词器并使之足够灵活，以便将其应用于多种类型和样式的输入。

这里使用的关键方法是 `Regex` 类的 `Split` 实例方法。该方法的返回值是一个字符串数组，其中的元素包括 `source` 字符串（本列中是 `equation`）的每个单独的标记。

注意静态方法 `Split` 允许使用 `RegexOptions` 枚举值，而实例方法允许定义起始位置和出现的最大匹配数量。这可能有助于你选择静态方法或实例方法。

7.4.4　参考

MSDN 文档中的“.NET Framework 正则表达式”主题。

7.5　返回匹配所在的整行内容

7.5.1　问题

你有一个包含多行内容的字符串或文件。当在一行上找到特写的字符模式时，你希望返回这一整行，而不仅仅是匹配的文本。

7.5.2　解决方案

使用 `StreamReader.ReadLine` 方法获得对其运行正则表达式的文件中的每一行，如例 7-4

所示。

例 7-4：返回匹配所在的整行内容

```
public static List<string> GetLines(string source, string pattern, bool isFileName)
{
    List<string> matchedLines = new List<string>();

    // 如果这是一个文件,获得整个文件的文本
    if (isFileName)
    {
        using (FileStream FS = new FileStream(source, FileMode.Open,
               FileAccess.Read, FileShare.Read))
        {
            using (StreamReader SR = new StreamReader(FS))
            {
                Regex RE = new Regex(pattern, RegexOptions.Multiline);
                string text = "";
                while (text != null)
                {
                    text = SR.ReadLine();
                    if (text != null)
                    {
                        // 对字符串中的每一行运行正则表达式
                        if (RE.IsMatch(text))
                        {
                            // 如果找到一个匹配,获得整行
                            matchedLines.Add(text);
                        }
                    }
                }
            }
        }
    }
    else
    {
        // 在整个字符串上运行一次正则表达式
        Regex RE = new Regex(pattern, RegexOptions.Multiline);
        MatchCollection theMatches = RE.Matches(source);

        // 使用这些变量记住添加到matchedLines的最后一行
        // 以便不会添加重复的行
        int lastLineStartPos = -1;
        int lastLineEndPos = -1;

        // 获得每个匹配所在的行
        foreach (Match m in theMatches)
        {
            int lineStartPos = GetBeginningOfLine(source, m.Index);
            int lineEndPos = GetEndOfLine(source, (m.Index + m.Length - 1));

            // 如果这不是一个重复行,添加它
            if (lastLineStartPos != lineStartPos &&
                lastLineEndPos != lineEndPos)
            {
                string line = source.Substring(lineStartPos,
```

```
                            lineEndPos - lineStartPos);
                matchedLines.Add(line);

                // 重置行位置
                lastLineStartPos = lineStartPos;
                lastLineEndPos = lineEndPos;
            }
        }
    }
    return (matchedLines);
}

public static int GetBeginningOfLine(string text, int startPointOfMatch)
{
    if (startPointOfMatch > 0)
    {
       --startPointOfMatch;
    }

    if (startPointOfMatch >= 0 && startPointOfMatch < text?.Length)
    {
       // 向左移动直到找到第一个\n字符
       for (int index = startPointOfMatch; index >= 0; index--)
       {
           if (text?[index] == '\n')
           {

               return (index + 1);
           }
       }

       return (0);
    }

    return (startPointOfMatch);
}

public static int GetEndOfLine(string text, int endPointOfMatch)
{
    if (endPointOfMatch >= 0 && endPointOfMatch < text?.Length)
    {
       // 向右移动直到找到第一个\n字符
       for (int index = endPointOfMatch; index < text.Length; index++)
       {
           if (text?[index] == '\n')
           {
               return (index);
           }
       }

       return (text.Length);
   }

   return (endPointOfMatch);
}
```

下面的方法显示了如何利用文件名或字符串调用 GetLines 方法。

```
public static void TestGetLine()
{
    // 获得文件TestFile.txt中的每一行并作为单独的字符串
    Console.WriteLine();
    List<string> lines = GetLines(@"C:\TestFile.txt", "Line", true);
    foreach (string s in lines)
        Console.WriteLine($"MatchedLine: {s}");

    // 获得给定字符串是匹配文本Line的所有行
    Console.WriteLine();
    lines = GetLines("Line1\r\nLine2\r\nLine3\nLine4", "Line", false);
    foreach (string s in lines)
        Console.WriteLine($"MatchedLine: {s}");
}
```

7.5.3 讨论

GetLines 方法接受下面三个参数。

- source

  在其中查找模式的字符串或文件名。

- pattern

  应用于 *source* 字符串的正则表达式模式。

- isFileName

  如果 *source* 是文件名，则传入 true；如果 *source* 是字符串，则传入 false。

该方法返回一个 List<string>，其中包含找到正则表达式匹配的每一行的字符串。

GetLines 方法可以获得字符串或文件内出现匹配的行。当对一个文件运行正则表达式时，该文件的名称传入 GetLines 方法中的 *source* 参数中（当 *isFileName* 等于 true 时），就会打开并逐行读取该文件。对每一行运行正则表达式，如果找到一个匹配，就把该行存储在 matchedLines List<string> 中。使用 StreamReader 对象的 ReadLine 方法，无需确定每一行开始和结束于何处。确定字符串中某一行开始和结束于何处需要一些工作，下面将会看到这一点。

对传入 GetLines 方法中 source 参数的字符串运行正则表达式（当 *isFileName* 等于 false 时）将会产生一个 MatchCollection。该集合中的每个 Match 对象用于获得 *source* 字符串中匹配所在的行。获得行的方法如下：从 *source* 字符串中匹配的第一个字符所在位置开始，向左移动一个字符，直至找到一个 \n 字符或者找到 *source* 字符串的开头（可以在 GetBeginningOfLine 方法中找到这段代码）。这将指示行的开头，它位于变量 lineStartPos 中。接下来，找到行的结尾，其方法如下：首先从 *source* 字符串中匹配的最后一个字符所在位置开始，并向右移动，直至找到一个 \n 字符或者找到 *source* 字符串的结尾（可以在 GetEndOfLine 方法中找到这段代码）。这个结尾位置位于 lineEndPos 变量中。lineStartPos 和 lineEndPos 之间的所有文本将是在其中找到匹配的行。将所有这些行添加到 matchedLines List<string> 中并返回调用者。

可以利用 GetLines 方法所做的一件有趣的事件是：在该方法的 pattern 参数中传入字符串 "\n"。这种技巧实际上会返回字符串或文件的每一行，作为 List<string> 中的字符串。尽管这个方法可用于内嵌 CRLF 字符的字符串，但是不能用于从文件中读出来的文本。原因是前面的 GetLines 方法中的 ReadLine 方法会去除 CRLF 字符。要修正这个问题，可以在 GetLines 方法中执行匹配时，简单地把这些字符添加回去。

要解决此问题，我们可以简单地将这些字符添加回去，因为我们在 GetLines 方法中执行匹配。

```
// 增加CRLF字符是有必要的
// 因为ReadLine()移除了这些字符
if (RE.IsMatch(text + Environment.NewLine))
```

最后，要注意如果在一行中找到多个匹配，会把每个匹配行添加到 List<string> 中。

当把换行符添加回文本时要小心。如果你仅在 Windows 系统上使用和处理文本，将不会有任何问题。然而，如果你使用其他系统或者混用多个系统，就需要确保添加了正确的换行符；也就是说，对于 Unix 和 OS X，仅使用换行符（\n）。

7.5.4 参考

MSDN 文档中的“.NET Framework 正则达式”“FileStream 类”和“StreamReader 类”主题。

7.6 找到特定次数的匹配

7.6.1 问题

你需要找到字符串内出现的特定次数的匹配。例如，你希望找到一个单词第三次出现的位置或者一个社会安全号码第二次出现的位置。此外，你还可能需要在字符串中找到一个单词每隔三次出现的位置。

7.6.2 解决方案

为了找到字符串中特定出现次数的匹配，只需对 Regex.Matches 返回的数组使用下标即可，代码如下所示。

```
public static Match FindOccurrenceOf(string source, string pattern,
                                     int occurrence)
{
    if (occurrence < 1)
    {
        throw (new ArgumentException("Cannot be less than 1",
                                     nameof(occurrence)));
    }
```

```
        // 使occurrence从0计数
        --occurrence;

        // 对source字符串运行一次正则表达式
        Regex RE = new Regex(pattern, RegexOptions.Multiline);
        MatchCollection theMatches = RE.Matches(source);

        if (occurrence >= theMatches.Count)
        {
            return (null);
        }
        else
        {
            return (theMatches[occurrence]);
        }
    }
```

为了在字符串中找到每个特定出现次数的匹配，可以实时构建一个 List<Match>。

```
    public static List<Match> FindEachOccurrenceOf(string source, string pattern,
                                                   int occurrence)
    {
        if (occurrence < 1)
        {
            throw (new ArgumentException("Cannot be less than 1",
                                         nameof(occurrence)));
        }

        List<Match> occurrences = new List<Match>();

        // 对source字符串运行一次正则表达式
        Regex RE = new Regex(pattern, RegexOptions.Multiline);
        MatchCollection theMatches = RE.Matches(source);

        for (int index = (occurrence - 1); index < theMatches.Count;
             index += occurrence)
        {
            occurrences.Add(theMatches[index]);
        }

        return (occurrences);
    }
```

下面的方法展示了如何调用前面两个方法。

```
    public static void TestOccurrencesOf()
    {
        Match matchResult = FindOccurrenceOf
                            ("one two three one two three one two three one"
                             + " two three one two three one two three", "two", 2);
        Console.WriteLine($"{matchResult?.ToString()}\t{matchResult?.Index}");

        Console.WriteLine();
        List<Match> results = FindEachOccurrenceOf
                              ("one one two three one two three one "
                               + " two three one two three", "one", 2);
```

```
        foreach (Match m in results)
            Console.WriteLine($"{m.ToString()}\t{m.Index}");
    }
```

7.6.3　讨论

本范例包含两个类似但截然不同的方法。第一个方法 `FindOccurrenceOf` 返回一个特定出现次数的正则表达式匹配。通过 `occurrence` 参数将你想要找到的出现次数传入该方法中。如果特定次数的匹配不存在（例如，你要求找到第二次出现的匹配，但是只存在一个匹配），就会从该方法返回 `null`。因此，在使用该方法返回对象之前应该确定它不是 `null`。如果存在特定的匹配，就会返回存有该次匹配信息的 `Match` 对象。

本范例中的第二个方法 `FindEachOccurrenceOf` 的工作方式类似于 `FindOccurrenceOf` 方法，只不过它将继续查找正则表达式的特定次数匹配，直至到达字符串的末尾。例如，如果你要求找到第二次出现的匹配，该方法将返回 0 个、1 个或多个 `Match` 对象的 `List<Match>`。`Match` 对象将对应于第二次、第四次、第六次和第八次出现的匹配，依此类推，直至到达字符串的末尾。

7.6.4　参考

MSDN 文档中的“.NET Framework 正则表达式”和“`ArrayList` 类”主题。

7.7　使用常见模式

7.7.1　问题

你需要一个快速列表，可从中选择匹配标准数据项的正则表达式模式。这些标准数据项可以是社会安全号码、邮政编码、只包含字符的单词、包含字母和数字的单词、电子邮件地址、URL、日期，也可以是各种商业应用程序中使用的其他可能的数据项。

这些模式可用于确保用户输入正确的数据并且格式标准。也可以把这些模式用作额外的安全措施，阻止黑客通过输入怪异的或畸形的数据（例如，SQL 注入或跨站脚本攻击）试图损坏你的代码。注意，这些正则表达式并不是阻止针对系统的所有攻击的银弹；确切地讲，它们只是附加的防御层。

7.7.2　解决方案

- 只匹配字母数字字符以及字符 `-`、`+`、`.` 和任何空白：

```
^([\w\.\+\-]|\s)*$
```

> 在字符类（封闭在方括号 [和] 内的正则表达式）中要小心使用 -（连字符）。该字符也用于指定字符的范围，比如 `a-z` 表示从 a 到 z（含 a 和 z）。如果想使用字面量 - 字符，就要用 \ 对它进行转义，或者把它放在表达式的末尾，如下一个示例所示。

- 只匹配字母数字字符以及字符 -、+、. 和任何空白，同时规定至少有其中的 1 个字符，并且不超过 10 个字符：
  ```
  ^([\w\.\+\-]|\s){1,10}$
  ```
- 匹配人名，最多 55 个字符：
  ```
  ^[a-zA-Z\'\-\s]{1,55}$
  ```
- 匹配正整数或负整数：
  ```
  ^(\+|\-)?\d+$
  ```
- 只匹配正浮点数或负浮点数；该模式不匹配整数：
  ```
  ^(\+|\-)?(\d*\.\d+)$
  ```
- 匹配可以具有正、负值的浮点数或整数：
  ```
  ^(\+|\-)?(\d*\.)?\d+$
  ```
- 匹配 ##/##/#### 形式的日期，其中日和月可以是 1 位或 2 位值，年只能是 4 位值：
  ```
  ^\d{1,2}\/\d{1,2}\/\d{4}$
  ```
- 验证输入是否是 ###-##-#### 形式的社会安全号码：
  ```
  ^\d{3}-\d{2}-\d{4}$
  ```
- 匹配 IPv4 地址：
  ```
  ^([0-2]?[0-9]?[0-9]\.){3}[0-2]?[0-9]?[0-9]$
  ```
- 验证电子邮件地址是否具有 name@address 形式，其中 address 不是一个 IP 地址：
  ```
  ^[A-Za-z0-9_\-\.]+@(([A-Za-z0-9\-])+\.)+([A-Za-z\-])+$
  ```
- 验证电子邮件地址是否具有 name@address 形式，其中 address 是一个 IP 地址：
  ```
  ^[A-Za-z0-9_\-\.]+@([0-2]?[0-9]?[0-9]\.){3}[0-2]?[0-9]?[0-9]$
  ```
- 匹配或验证使用 HTTP、HTTPS 或 FTP 协议的 URL。注意，该正则表达式不匹配相对 URL：
  ```
  ^(http|https|ftp)\://[a-zA-Z0-9\-\.]+\.[a-zA-Z]{2,3}(:[a-zA-Z0-9]*)?/?
  ([a-zA-Z0-9\-\._\?\,\'/\\\+&%\$#\=~])*$
  ```
- 只匹配美元金额，带有可选的 $ 以及 + 或 - 前缀字符（注意，可以添加任意数量的小数位）：
  ```
  ^\$?[+-]?[\d,]*(\.\d*)?$
  ```

这类似于前一个正则表达式，只不过允许的小数位不能超过 2 位：

```
^\$?[+-]?[\d,]*\.?\d{0,2}$
```

- 匹配作为 4 组 4 位数字输入的信用卡号，可以用空格或 - 字符隔开这 4 组数字，也可以连在一起：
  ```
  ^((\d{4}[- ]?){3}\d{4})$
  ```
- 匹配作为 5 位数字输入的邮政编码，带有可选的 4 位数字的扩展：
  ```
  ^\d{5}(-\d{4})?$
  ```
- 匹配一个北美的电话号码，带有可选的区号并且可以在电话号码中使用 - 字符，没有分

机号：

```
^(\(?[0-9]{3}\)?)?\-?[0-9]{3}\-?[0-9]{4}$
```

- 匹配类似于前一个正则表达式的电话号码，但是允许可选的 5 位分机号，前缀为 `ext` 或 `extension`：

```
^(\(?[0-9]{3}\)?)?\-?[0-9]{3}\-?[0-9]{4}(\s*ext(ension)?[0-9]{5})?$
```

- 匹配以驱动器字母开头的完整路径，并且可选择地匹配带有 3 字符扩展名的文件名（注意，不允许任何表示向上移动目录层次结构的 `..` 字符，也不允许点（`.`）后面跟着扩展名的目录名）：

```
^[a-zA-Z]:[\\/]([_a-zA-Z0-9]+[\\/]?)*([_a-zA-Z0-9]+\.[_a-zA-Z0-9]{0,3})?$
```

- 验证输入密码字符串是否匹配用于输入密码的一些指定规则（即密码长度在 6~25 个字符之间，并且包含字母和数字字符）：

```
^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{6,25}$
```

- 确定用户是否输入了任何恶意字符。注意，该正则表达式不仅会阻止所有恶意输入，还会阻止一些合法的输入，比如包含单引号的姓氏：

```
^([^\)\(\<\>\"\'\%\&\+\;][(-{2})])*$
```

- 从 XHTML、HTML 或 XML 字符串中提取标签。该正则表达式将返回开始标签和结束标签，包括标签的任何属性。注意，需要用想要查找的真实标签名称替换 *`TAGNAME`*：

```
<TAGNAME.*?>(.*?)</TAGNAME>
```

- 从代码中提取注释行。下面的正则表达式从 Web 页面中提取 HTML 注释。这可用于确定在投入实际应用前是否需要从代码库中删除任何泄露了敏感信息的 HTML 注释：

```
<!--.*?-->
```

- 匹配 C# 单行注释：

```
//.*$
```

- 匹配 C# 多行注释：

```
/\*.*?\*/
```

虽然前述的 4 个正则表达式非常适合于查找标签和注释，但它们不是万无一失的。为了准确查找所有的标签和注释，需要使用针对目标语言的完整解析器。

7.7.3 讨论

正则表达式可以有效地查找特定的信息，并且它们具有广泛的使用范围。许多应用程序使用它们来查找更大范围文本内的特定信息，并且过滤掉有害的输入。在加强应用程序的安全以及防止黑客试图用精心构造的输入来获得互联网或局域网上机器的访问权限方面，过滤操作非常有用。使用正则表达式只允许将良好的输入传递给应用程序，可以减少发生多种攻击的可能性，比如 SQL 注入或跨站脚本攻击。

本范例中展示的正则表达式只提供了关于可以利用它们完成什么任务的简单介绍。可以轻松地修改这些表达式来满足你的需求。例如，如下表达式只允许输入 1~10 之间的数字字符以及少数几个符号。

```
^([\w\.\+\-]|\s){1,10}$
```

通过把正则表达式的 `{1,10}` 部分更改为 `{0,200}`，该正则表达式现在将匹配空白条目或者最多包含 200 个字符的指定符号的条目。

注意在表达式开头使用的 `^` 字符和在表达式末尾使用的 `$` 字符。这些字符从文本开始处开始匹配，并一直匹配到文本末尾。添加这些字符将强制正则表达式匹配整个字符串或者一点也不匹配。通过删除这些字符，可以查找大块文本内的特定文本。例如，下面的正则表达式仅仅匹配其中只包含美国邮政编码（不能有前导或尾随空格）的字符串。

```
^\d{5}(-\d{4})?$
```

下面这个版本只匹配具有前导或尾随空格的邮政编码（注意，在表达式的开头和末尾添加了 `\s*`）。

```
^\s*\d{5}(-\d{4})?\s*$
```

不过，下面这个经过修改的表达式匹配字符串内任意位置找到的邮政编码（包括只包含邮政编码的字符串）。

```
\d{5}(-\d{4})?
```

使用本范例中介绍的正则表达式，并修改它们以满足你的需求。

7.7.4　参考

Michael Fitzgerald 著的《学习正则表达式》以及 Jan Goyvaerts 和 Steven Levithan 合著的《正则表达式经典实例》。这两本英文原书都是由 O’Reilly 出版的。

第 8 章

文件系统I/O

8.0 简介

本章处理一些与文件系统有关的主题，如基于目录或文件夹的编程任务。本章也提到了下面这些文件系统 I/O（输入 / 输出）中更高级的主题。

- 锁定文件的一部分
- 监控某些文件系统行为
- 文件中的版本信息
- 文件压缩

各种文件和目录 I/O 技术的使用贯穿在各个范例中，展示了如何执行创建、打开、删除、读取和写入文件和目录等任务。这些基本知识有助于用户理解其他文件 I/O 范例以及如何根据用户目标进行修改。

多个范例已更新以使用 `async` 和 `await` 运算符帮助减轻处理文件系统、网络或文件 I/O 时通常会遇到的延迟。使用 `async` 和 `await` 通过允许 I/O 操作进行但不会像通常那样在完成前阻塞调用线程，从而提高了代码的整体响应能力。

除非另有说明，在使用本章中的代码片段或方法的任何程序中，都需要下列 `using` 语句。

```
using System;
using System.IO;
```

8.1 使用通配符查找目录和文件

8.1.1 问题

你正在尝试在当前文件系统中查找可能存在也可能不存在的一个或多个特定文件或目录。搜索可能需要使用通配符，以便扩大搜索。例如，搜索一个文件系统中所有用户模式的转储文件。这些文件都具有 .dmp 扩展名。

8.1.2 解决方案

获得这一信息有几种方法。前三种方法返回一个包含每个数据项的完整路径的字符串数组。接下来的三种方法返回一个封装了一个目录、一个文件或者全部两者的对象。

`Directory` 类上的静态 `GetFileSystemEntries` 方法返回一个包含单个目录内所有文件和目录名的字符串数组，如下所示。

```
public static void DisplayFilesAndSubDirectories(string path)
{
    if (string.IsNullOrWhiteSpace(path))
        throw new ArgumentNullException(nameof(path));

    string[] items = Directory.GetFileSystemEntries(path);
    Array.ForEach(items, item =>
    {
        Console.WriteLine(item);
    });
}
```

`Directory` 类上的静态 `GetDirectories` 方法返回一个包含单个目录内所有目录名的字符串数组。下面的 `DisplaySubDirectories` 方法展示了你可以如何使用它。

```
public static void DisplaySubDirectories(string path)
{
    if (string.IsNullOrWhiteSpace(path))
        throw new ArgumentNullException(nameof(path));

    string[] items = Directory.GetDirectories(path);
    Array.ForEach(items, item =>
    {
        Console.WriteLine(item);
    });
}
```

`Directory` 类的静态 `GetFiles` 方法返回一个包含单个目录内所有文件名的字符串数组。下列方法与 `DisplaySubDirectories` 非常类似，但调用的是 `Directory.GetFiles` 而不是 `Directory.GetDirectories`。

```
public static void DisplayFiles(string path)
{
    if (string.IsNullOrWhiteSpace(path))
        throw new ArgumentNullException(nameof(path));
```

```
        string[] items = Directory.GetFiles(path);
        Array.ForEach(items, item =>
        {
            Console.WriteLine(item);
        });
    }
```

接下来的两个方法返回一个对象而不是仅仅返回一个字符串。DirectoryInfo 对象的 GetFileSystemInfos 方法返回一个代表单个目录内目录和文件的 FileSystemInfo 对象（也就是 DirectoryInfo 和 FileInfo 对象）构成的强类型数组。下面的示例调用 GetFileSystemInfos 方法以获取一个由代表某个特定目录中所有项的 FileSystemInfo 对象构成的数组，然后在控制台窗口中列出一个关于 FileSystemInfo 的显示信息字符串。显示信息由 FileSystemInfo 上的扩展方法 ToDisplayString 生成。

```
public static void DisplayDirectoryContents(string path)
{
    if (string.IsNullOrWhiteSpace(path))
        throw new ArgumentNullException(nameof(path));

    DirectoryInfo mainDir = new DirectoryInfo(path);
    var fileSystemDisplayInfos =
        (from fsi in mainDir.GetFileSystemInfos()
        where fsi is FileSystemInfo || fsi is DirectoryInfo
        select fsi.ToDisplayString()).ToArray();

    Array.ForEach(fileSystemDisplayInfos, s =>
    {
        Console.WriteLine(s);
    });
}

public static string ToDisplayString(this FileSystemInfo fileSystemInfo)
{
    string type = fileSystemInfo.GetType().ToString();
    if (fileSystemInfo is DirectoryInfo)
        type = "DIRECTORY";
    else if (fileSystemInfo is FileInfo)
        type = "FILE";
    return $"{type}: {fileSystemInfo.Name}";
}
```

这段代码的输出如下所示。

```
DIRECTORY: MyNestedTempDir
DIRECTORY: MyNestedTempDirPattern
FILE: MyTempFile.PDB
FILE: MyTempFile.TXT
```

DirectoryInfo 对象的 GetDirectories 实例方法仅返回一个代表单个目录下子目录的 DirectoryInfo 对象的数组。例如，下列代码调用 GetDirectories 方法以获得一个 DirectoryInfo 对象的数组，然后将每个对象的 Name 属性显示到控制台窗口。

```
public static void DisplayDirectoriesFromInfo(string path)
{
    if (string.IsNullOrWhiteSpace(path))
        throw new ArgumentNullException(nameof(path));

    DirectoryInfo mainDir = new DirectoryInfo(path);
    DirectoryInfo[] items = mainDir.GetDirectories();
    Array.ForEach(items, item =>
    {
        Console.WriteLine($"DIRECTORY: {item.Name}");
    });
}
```

DirectoryInfo 对象的 GetFiles 实例方法仅返回一个代表单个目录下文件的 FileInfo 对象的数组。例如，以下代码调用 GetFiles 方法以获得一个 FileInfo 对象的数组，然后将每个对象的 Name 属性显示到控制台窗口。

```
public static void DisplayFilesFromInfo(string path)
{
    if (string.IsNullOrWhiteSpace(path))
        throw new ArgumentNullException(nameof(path));

    DirectoryInfo mainDir = new DirectoryInfo(path);
    FileInfo[] items = mainDir.GetFiles();
    Array.ForEach(items, item =>
    {
        Console.WriteLine($"FILE: {item.Name}");
    });
}
```

Directory 类上的静态 GetFileSystemEntries 方法返回单个目录下所有匹配模式的文件和目录。

```
public static void DisplayFilesWithPattern(string path, string pattern)
{
    if (string.IsNullOrWhiteSpace(path))
        throw new ArgumentNullException(nameof(path));
    if (string.IsNullOrWhiteSpace(pattern))
        throw new ArgumentNullException(nameof(pattern));

    string[] items = Directory.GetFileSystemEntries(path, pattern);
    Array.ForEach(items, item =>
    {
        Console.WriteLine(item);
    });
}
```

Directory 类上的静态 GetDirectories 方法仅返回单个目录下所有匹配模式的目录。

```
public static void DisplayDirectoriesWithPattern(string path, string pattern)
{
    if (string.IsNullOrWhiteSpace(path))
        throw new ArgumentNullException(nameof(path));
    if (string.IsNullOrWhiteSpace(pattern))
        throw new ArgumentNullException(nameof(pattern));
```

```
        string[] items = Directory.GetDirectories(path, pattern);
        Array.ForEach(items, item =>
        {
            Console.WriteLine(item);
        });
    }
```

Directory 类上的静态 GetFiles 方法仅返回单个目录下所有匹配模式的文件。

```
    public static void DisplayFilesWithGetFiles(string path, string pattern)
    {
        if (string.IsNullOrWhiteSpace(path))
            throw new ArgumentNullException(nameof(path));
        if (string.IsNullOrWhiteSpace(pattern))
            throw new ArgumentNullException(nameof(pattern));

        string[] items = Directory.GetFiles(path, pattern);
        Array.ForEach(items, item =>
        {
            Console.WriteLine(item);
        });
    }
```

接下来的三个方法返回一个对象而不是仅仅返回一个字符串。第一个实例方法是 GetFileSystemInfos，它返回单个目录内匹配模式的目录和文件。

```
    public static void DisplayDirectoryContentsWithPattern(string path,
        string pattern)
    {
        if (string.IsNullOrWhiteSpace(path))
            throw new ArgumentNullException(nameof(path));
        if (string.IsNullOrWhiteSpace(pattern))
            throw new ArgumentNullException(nameof(pattern));

        DirectoryInfo mainDir = new DirectoryInfo(path);
        var fileSystemDisplayInfos =
            (from fsi in mainDir.GetFileSystemInfos(pattern)
            where fsi is FileSystemInfo || fsi is DirectoryInfo
            select fsi.ToDisplayString()).ToArray();

        Array.ForEach(fileSystemDisplayInfos, s =>
        {
            Console.WriteLine(s);
        });
    }
```

GetDirectories 实例方法仅返回单个目录内匹配模式的目录（包含在 DirectoryInfo 对象中）。

```
    public static void DisplayDirectoriesWithPatternFromInfo(string path,
        string pattern)
    {
        if (string.IsNullOrWhiteSpace(path))
            throw new ArgumentNullException(nameof(path));
```

```
        if (string.IsNullOrWhiteSpace(pattern))
            throw new ArgumentNullException(nameof(pattern));

        DirectoryInfo mainDir = new DirectoryInfo(path);
        DirectoryInfo[] items = mainDir.GetDirectories(pattern);
        Array.ForEach(items, item =>
        {
            Console.WriteLine($"DIRECTORY: {item.Name}");
        });
    }
```

GetFiles 实例方法仅返回单个目录内匹配模式的文件信息（包含在 FileInfo 对象中）。

```
    public static void DisplayFilesWithInstanceGetFiles(string path, string pattern)
    {
        if (string.IsNullOrWhiteSpace(path))
            throw new ArgumentNullException(nameof(path));
        if (string.IsNullOrWhiteSpace(pattern))
            throw new ArgumentNullException(nameof(pattern));

        DirectoryInfo mainDir = new DirectoryInfo(path);
        FileInfo[] items = mainDir.GetFiles(pattern);
        Array.ForEach(items, item =>
        {
            Console.WriteLine($"FILE: {item.Name}");
        });
    }
```

8.1.3 讨论

如果只需要一个包含目录和文件路径的字符串数组，那么可以使用静态方法 Directory.GetFileSystemEntries。返回的字符串数组不包含任何关于一个单独元素是目录还是文件的信息。每个字符串元素包含指向特定路径内包含的某个目录或文件的完整路径。

为了快速、简易地辨认出目录和文件，可使用 Directory.GetDirectories 和 Directory.GetFiles 静态方法。这些方法返回目录名和文件名的数组。这些方法返回一个字符串对象的数组。每个元素包含目录或文件的完整路径。

如果用户不需要关于所返回的目录或文件的任何其他信息，或者如果用户需要所返回的其中一个文件的更多信息，那么返回一个字符串就可以了。使用静态方法获得文件名列表并且仅获取用户所需的 FileInfo 比构造目录下所有的 FileInfo 更有效率，就如实例方法将会做的那样。如果必须访问每个文件的属性、长度或时间，那么应当考虑使用获取 FileInfo 细节的实例方法。

实例方法 GetFileSystemInfos 返回一个强类型 FileSystemInfo 对象构成的数组（FileSystemInfo 对象是 DirectoryInfo 和 FileInfo 对象的基类）。因此，使用 is 或 as 关键字就可以测试返回的类型是一个 DirectoryInfo 对象还是一个 FileInfo 对象。一旦知道了该对象实际是哪一个子类，就可以将其类型转换到子类并开始使用它。

要仅获得 DirectoryInfo 对象，可使用重载的 GetDirectories 实例方法。要仅获得 FileInfo 对象，可使用重载的 GetFiles 实例方法。这些方法分别返回一个 DirectoryInfo

和 FileInfo 对象构成的数组，每个数组的元素都封装了一个目录或文件。

当从 GetFiles 或 GetFileSystemInfos 中过滤结果时，可提供的模式有以下这些需要注意的行为。

- 模式不能包含任何 InvalidPathChars,并且不能使用在目录结构中回到上一个级别的 ..。
- 返回的数组中项的顺序不能确定，但是可以使用 Sort 或者在一个查询中对结果排序。
- 当扩展名恰好是 3 个字符时，行为有些不同：模式将与任何在扩展名中包括这 3 个字符的文件匹配。
- *.htm 返回带有扩展名为 .htm、.html、.htma 等的文件。
- 当一个扩展名少于或多于 3 字符时，模式将执行精确匹配。
- *.cs 仅返回扩展名为 .cs 的文件。

8.1.4 参考

MSDN 文档中的“DirectoryInfo 类”“FileInfo 类”和“FileSystemInfo 类”主题。

8.2 获取目录树

8.2.1 问题

你需要获得一棵目录树，可能包含文件名，从目录层次内的任意一点扩展开来。此外，返回的每个目录或文件必须具有封装该项的对象形式。这样允许你在返回的对象上执行操作，例如删除文件、重命名文件或者检查 / 修改其属性。最后，你可能需要根据一个模式搜索指定子集的能力，例如仅查找具有 .pdb 扩展名的文件。

8.2.2 解决方案

通过调用 GetFileSystemInfos 实例方法，可以获取从任何起始点沿目录层次向下的所有文件和目录所构成的一个可枚举列表，代码如下所示。

```
public static IEnumerable<FileSystemInfo> GetAllFilesAndDirectories(string dir)
{
    if (string.IsNullOrWhiteSpace(dir))
        throw new ArgumentNullException(nameof(dir));

    DirectoryInfo dirInfo = new DirectoryInfo(dir);
    Stack<FileSystemInfo> stack = new Stack<FileSystemInfo>();

    stack.Push(dirInfo);
    while (dirInfo != null || stack.Count > 0)
    {
        FileSystemInfo fileSystemInfo = stack.Pop();
        DirectoryInfo subDirectoryInfo = fileSystemInfo as DirectoryInfo;
        if (subDirectoryInfo != null)
        {
            yield return subDirectoryInfo;
            foreach (FileSystemInfo fsi in subDirectoryInfo.GetFileSystemInfos())
```

```
                stack.Push(fsi);
            dirInfo = subDirectoryInfo;
        }
        else
        {
            yield return fileSystemInfo;
            dirInfo = null;
        }
    }
}
```

要显示文件和目录获取的结果，可使用下列查询。

```
public static void DisplayAllFilesAndDirectories(string dir)
{
    if (string.IsNullOrWhiteSpace(dir))
        throw new ArgumentNullException(nameof(dir));

    var strings = (from fileSystemInfo in GetAllFilesAndDirectories(dir)
                   select fileSystemInfo.ToDisplayString()).ToArray();

    Array.ForEach(strings, s => { Console.WriteLine(s); });
}
```

因为结果是可查询的，因此不必获取关于所有文件和目录的信息。下列查询使用一个不区分大小写的比较来获得存在目录中所有包含 Chapter 1 且扩展名为 .pdb 的文件列表。

```
public static void DisplayAllFilesWithExtension(string dir, string extension)
{
    if (string.IsNullOrWhiteSpace(dir))
        throw new ArgumentNullException(nameof(dir));
    if (string.IsNullOrWhiteSpace(extension))
        throw new ArgumentNullException(nameof(extension));

    var strings = (from fileSystemInfo in GetAllFilesAndDirectories(dir)
                   where fileSystemInfo is FileInfo &&
                           fileSystemInfo.FullName.Contains("Chapter 1") &&
                           (string.Compare(fileSystemInfo.Extension, extension,
                                 StringComparison.OrdinalIgnoreCase) == 0)
                   select fileSystemInfo.ToDisplayString()).ToArray();

    Array.ForEach(strings, s => { Console.WriteLine(s); });
}
```

8.2.3 讨论

要获得一棵代表目录和它所包含文件的树，可以在一个方法中使用递归式迭代器，如下所示。

```
public static IEnumerable<FileSystemInfo> GetAllFilesAndDirectoriesWithRecursion(
    string dir)
{
    if (string.IsNullOrWhiteSpace(dir))
        throw new ArgumentNullException(nameof(dir));
```

```
    DirectoryInfo dirInfo = new DirectoryInfo(dir);
    FileSystemInfo[] fileSystemInfos = dirInfo.GetFileSystemInfos();
    foreach (FileSystemInfo fileSystemInfo in fileSystemInfos)
    {
        yield return fileSystemInfo;
        if (fileSystemInfo is DirectoryInfo)
        {
            foreach (FileSystemInfo fsi in
GetAllFilesAndDirectoriesWithRecursion(fileSystemInfo.FullName))
                yield return fsi;
        }
    }
}

public static void DisplayAllFilesAndDirectoriesWithRecursion(string dir)
{
    if (string.IsNullOrWhiteSpace(dir))
        throw new ArgumentNullException(nameof(dir));

    var strings = (from fileSystemInfo in
                   GetAllFilesAndDirectoriesWithRecursion(dir)
                   select fileSystemInfo.ToDisplayString()).ToArray();

    Array.ForEach(strings, s => { Console.WriteLine(s); });
}
```

这与解决方案中代码的主要区别是，它使用递归式迭代器，而解决方案使用迭代式迭代器和一个显式的栈。

> 你不会想使用递归式迭代方法，因为其性能实际是 $O(n*d)$，其中 *n* 是 `FileSystemInfos` 的数目，而 *d* 是目录层次的深度，它一般等于 $\log n$。参见演示代码。

如果解决方案方法被分别重命名为 `GetAllFilesAndDirectoriesWithoutRecursion` 和 `DisplayAllFilesAndDirectoriesWithoutRecursion`，那么可以使用下列代码检查其性能。

```
string dir = Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles);

// 不使用递归显示所有文件
Stopwatch watch1 = Stopwatch.StartNew();
DisplayAllFilesAndDirectoriesWithoutRecursion(tempDir1);
watch1.Stop();
Console.WriteLine("*************************");

// 使用递归列出所有文件
Stopwatch watch2 = Stopwatch.StartNew();
DisplayAllFilesAndDirectoriesWithRecursion(tempDir1);
watch2.Stop();
Console.WriteLine("*************************");
Console.WriteLine(
    $"Non-Recursive method time elapsed {watch1.Elapsed.ToString()}");
Console.WriteLine($"Recursive method time elapsed {watch2.Elapsed.ToString()}");
```

不使用递归的方法如下所示。

```
public static void DisplayAllFilesAndDirectoriesWithoutRecursion(string dir)
{
    var strings = from fileSystemInfo in
                    GetAllFilesAndDirectoriesWithoutRecursion(dir)
                    select fileSystemInfo.ToDisplayString();

    foreach (string s in strings)
        Console.WriteLine(s);
}

public static void DisplayAllFilesWithExtensionWithoutRecursion(string dir,
    string extension)
{
    var strings = from fileSystemInfo in
                    GetAllFilesAndDirectoriesWithoutRecursion(dir)
                    where fileSystemInfo is FileInfo &&
                        fileSystemInfo.FullName.Contains("Chapter 1") &&
                        (string.Compare(fileSystemInfo.Extension, extension,
                                        StringComparison.OrdinalIgnoreCase) == 0)
                    select fileSystemInfo.ToDisplayString();

    foreach (string s in strings)
        Console.WriteLine(s);
}

public static IEnumerable<FileSystemInfo>
    GetAllFilesAndDirectoriesWithoutRecursion(
    string dir)
{
    DirectoryInfo dirInfo = new DirectoryInfo(dir);
    Stack<FileSystemInfo> stack = new Stack<FileSystemInfo>();

    stack.Push(dirInfo);
    while (dirInfo != null || stack.Count > 0)
    {
        FileSystemInfo fileSystemInfo = stack.Pop();
        DirectoryInfo subDirectoryInfo = fileSystemInfo as DirectoryInfo;
        if (subDirectoryInfo != null)
        {
            yield return subDirectoryInfo;
            foreach (FileSystemInfo fsi in subDirectoryInfo.GetFileSystemInfos())
                stack.Push(fsi);
            dirInfo = subDirectoryInfo;
        }
        else
        {
            yield return fileSystemInfo;
            dirInfo = null;
        }
    }
}
```

8.2.4 参考

MSDN 文档中的“DirectoryInfo 类”“FileInfo 类”和“FileSystemInfo 类”主题。

8.3 解析路径

8.3.1 问题

你需要分离一个路径的组成部分并把它们存放在单独的变量中。

8.3.2 解决方案

使用 Path 类的静态方法：

```
public static void DisplayPathParts(string path)
{
    if (string.IsNullOrWhiteSpace(path))
        throw new ArgumentNullException(nameof(path));

    string root = Path.GetPathRoot(path);
    string dirName = Path.GetDirectoryName(path);
    string fullFileName = Path.GetFileName(path);
    string fileExt = Path.GetExtension(path);
    string fileNameWithoutExt = Path.GetFileNameWithoutExtension(path);
    StringBuilder format = new StringBuilder();
    format.Append($"ParsePath of {path} breaks up into the following pieces:" +
        $"{Environment.NewLine}");
    format.Append($"\tRoot: {root}{Environment.NewLine}");
    format.Append($"\tDirectory Name: {dirName}{Environment.NewLine}");
    format.Append($"\tFull File Name: {fullFileName}{Environment.NewLine}");
    format.Append($"\tFile Extension: {fileExt}{Environment.NewLine}");
    format.Append($"\tFile Name Without Extension: {fileNameWithoutExt}" +
        $"{Environment.NewLine}");
    Console.WriteLine(format.ToString());
}
```

如果将字符串 C:\test\tempfile.txt 传递给该方法，那么输出如下所示。

```
ParsePath of C:\test\tempfile.txt breaks up into the following pieces:
        Root: C:\
        Directory Name: C:\test
        Full File Name: tempfile.txt
        File Extension: .txt
        File Name Without Extension: tempfile
```

8.3.3 讨论

Path 类包含可以用于解析某个给定路径的方法。与编写路径解析和文件名解析代码相比，使用这些类要简单得多，并且不容易出错。如果不使用这些类，在安全决策中手动解析例程中收集的信息时，也可能会把安全漏洞带入应用程序。用于解析路径的五个主要方法

是 GetPathRoot、GetDirectoryName、GetFileName、GetExtension 和 GetFileNameWithout-Extension。每种方法都只有一个参数 path，代表要解析的路径。

- GetPathRoot

  该方法返回路径的根目录。如果路径中没有根目录，例如使用一个相对路径时，该方法返回一个空字符串，而不是 null。

- GetDirectoryName

  该方法返回文件所在目录的完整路径。

- GetFileName

  该方法返回文件名，包含文件的扩展名。如果路径中没有提供文件名，那么该方法返回一个空字符串，而不是 null。

- GetExtension

  该方法返回文件的扩展名。如果没有提供文件的扩展名或者路径中不存在文件，那么该方法返回一个空字符串，而不是 null。

- GetFileNameWithoutExtension

  该方法返回不带文件名的根文件名。

要注意，这些方法并非实际确定驱动器、目录甚至文件是否存在于运行这些方法的系统中。这些方法是字符串解析器，如果为其中之一传递一个具有某种奇怪格式的字符串（例如 \\ZY:\foo)，那么它无论如何都将试图做它能够解析的工作。

```
ParsePath of \\ZY:\foo breaks up into the following pieces:
        Root: \\ZY:\foo
        Directory Name:
        Full File Name: foo
        File Extension:
        File Name Without Extension: foo
```

但是，如果路径中发现非法字符，那么这些方法将引发一个异常。

要想确定文件或目录是否存在，可使用静态方法 Directory.Exists 或 File.Exists。

8.3.4 参考

MSDN 文档中的“Path 类”主题。

8.4 启动并与控制台工具交互

8.4.1 问题

你有一个需要自动运行并且仅从标准输入流中获取输入的应用程序。你需要通过标准输入流发送命令来驱动此应用程序。

8.4.2 解决方案

假定我们需要使用 TIME /T 命令驱动 cmd.exe 应用程序显示当前时间。(可以直接通过命令行运行该命令，但用这种方法我们可以展示一种替代的方式去驱动一个响应标准输入的应用程序。)完成这一工作的方法是运行一个在标准输入流上查找输入的进程。这可以通过 Process 类的 StartInfo 属性完成，它是 ProcessStartInfo 类的一个实例。StartInfo 的字段可以控制新进程运行环境的许多细节，Process.Start 将会使用这些选项启动一个新线程。

首先，确保将 StartInfo.RedirectStandardInput 属性设置为 true。这一设置通知进程，它应当从标准输入中读取。然后，将 StartInfo.UseShellExecute 属性设置为 false，因为如果你打算让 shell 为你运行该进程，那么它会阻止重定向标准输入。

一旦完成这些，启动该进程并写入其标准输入流，如例 8-1 所示。

例 8-1：RunProcessToReadStandardInput 方法

```
public static void RunProcessToReadStandardInput()
{
    Process application = new Process();
    // 运行命令行shell
    application.StartInfo.FileName = @"cmd.exe";

    // 打开cmd.exe的命令扩展
    application.StartInfo.Arguments = "/E:ON";

    application.StartInfo.RedirectStandardInput = true;

    application.StartInfo.UseShellExecute = false;

    application.Start();

    StreamWriter input = application.StandardInput;
    // 运行命令以显示时间
    input.WriteLine("TIME /T");

    // 停止我们启动的应用程序
    input.WriteLine("exit");
}
```

8.4.3 讨论

重定向一个进程的输入流允许你以编程方式与某些应用程序和工具交互，否则需要额外的工具才能实现自动化。一旦输入被重定向，你就能够通过读取 Process.StandardInput 属性返回一个 StreamWriter，然后写入进程的标准输入流。获取了 StreamWriter 后，你就能够通过 WriteLine 调用将内容发送到进程，如前所示。

为了使用 StandardInput，必须将 StartInfo 属性的 RedirectStandardInput 属性指定为 true。否则，读取 StandardInput 属性会引发一个异常。

当 UseShellExecute 为 false 时，你仅能使用 Process 创建可执行的进程。正常情况

下，Process 类可用于在文件上执行操作，例如打印一份 Microsoft Word 文档。当 UseShellExecute 为 false 时的另一个差异是工作目录不会用于查找可执行文件，因此必须注意传递一个完整路径，或者将可执行文件所在的目录加入 PATH 环境变量。

8.4.4　参考

MSDN 文档中的“Process 类”“ProcessStartInfo 类”“RedirectStandardInput 属性”和“UseShellExecute 属性”主题。

8.5　锁定文件的一部分

8.5.1　问题

你需要从一个文件中部分读取或写入数据，并且希望确保在完成这一操作之前没有其他进程或线程能够访问、修改和删除该文件。

8.5.2　解决方案

锁定以防止其他进程在你使用文件时访问你的文件，可通过 FileStream 类的 Lock 方法完成。下列代码根据 fileName 参数生成一个文件并写入两行文本。然后使用 Lock 方法对整个文件加锁。当文件被锁定时，代码继续执行一些其他处理；当该代码返回时，文件被关闭，从而将其解锁。

```
public static async Task CreateLockedFileAsync(string fileName)
{
    if (string.IsNullOrWhiteSpace(fileName))
        throw new ArgumentNullException(nameof(fileName));

    FileStream fileStream = null;
    try
    {
        fileStream = new FileStream(fileName,
                FileMode.Create,
                FileAccess.ReadWrite,
                FileShare.ReadWrite, 4096, useAsync: true);

        using (StreamWriter writer = new StreamWriter(fileStream))
        {
            await writer.WriteLineAsync("The First Line");
            await writer.WriteLineAsync("The Second Line");
            await writer.FlushAsync();

            try
            {
                // 锁定整个文件
                fileStream.Lock(0, fileStream.Length);

                // 执行一些耗时的处理
                Thread.Sleep(1000);
```

```
            }
            finally
            {
                // 确保解锁文件
                // 如果一个进程终结时仍有文件的一部分被锁定，
                // 或者关闭一个带有锁定的文件，其行为是未定义的
                fileStream.Unlock(0, fileStream.Length);
            }

            await writer.WriteLineAsync("The Third Line");
            fileStream = null;
        }
    }
    finally
    {
        if (fileStream != null)
            fileStream.Dispose();
    }
}
```

在 CreateLockedFileAsync 中使用了 async 和 await 运算符。async 运算符允许你指明一个方法可以在特定位置暂停，await 运算符用来指定这些代码中的暂停点，这意味着编译器知道 async 方法无法继续执行到暂停点后，直到 await 指定的异步处理完成。在它等待时，调用方获得控制权。这有助于你的程序中调用者的线程不会阻塞，并且可以执行其他工作，但该方法的表现仍然像是被同步调用的。

8.5.3　讨论

如果在你的应用程序中打开了一个文件，并且将 FileStream.Open 调用的 FileShare 参数设置为 FileShare.ReadWrite 或 FileShare.Write，那么应用程序中的其他代码可以在你使用文件时查看或改变文件的内容。为了以更细的粒度处理文件访问，可使用 FileStream 对象的 Lock 方法，防止其他代码改写用户文件的部分或全部。一旦完成对文件加锁部分的操作，就可以调用 FileStream 对象的 Unlock 方法，从而允许用户应用程序中的其他代码向文件的该部分写入数据。

要锁定整个文件，可使用下列语法。

```
fileStream.Lock(0, fileStream.Length);
```

要锁定文件的一部分，可使用下列语法。

```
fileStream.Lock(4, fileStream.Length - 4);
```

这一行代码锁定除前四个字符外的整个文件。注意，可以锁定整个文件并且仍然能多次打开它，也能写入内容。

如果另一线程正在访问该文件，那么在调用 Write、Flush 或 Close 方法时，可能会看到引发一个 IOException。例如，下面的代码就容易产生这种异常。

```
public static async Task CreateLockedFileWithExceptionAsync(string fileName)
{

    FileStream fileStream = null;
    try
    {
        fileStream = new FileStream(fileName,
                FileMode.Create,
                FileAccess.ReadWrite,
                FileShare.ReadWrite, 4096, useAsync: true);
        using (StreamWriter streamWriter = new StreamWriter(fileStream))
        {
            await streamWriter.WriteLineAsync("The First Line");
            await streamWriter.WriteLineAsync("The Second Line");
            await streamWriter.FlushAsync();

            // 锁定整个文件
            fileStream.Lock(0, fileStream.Length);

            FileStream writeFileStream = null;
            try
            {
                writeFileStream = new FileStream(fileName,
                                            FileMode.Open,
                                            FileAccess.Write,
                                            FileShare.ReadWrite, 4096,
                                            useAsync: true);
                using (StreamWriter streamWriter2 =
                    new StreamWriter(writeFileStream))
                {
                    await streamWriter2.WriteAsync("foo ");
                    try
                    {
                        streamWriter2.Close(); // --> 此处会发生异常
                    }
                    catch
                    {
                        Console.WriteLine(
                        "The streamWriter2.Close call generated an exception.");
                    }
                    streamWriter.WriteLine("The Third Line");
                }
                writeFileStream = null;
            }
            finally
            {
                if (writeFileStream != null)
                    writeFileStream.Dispose();
            }
        }
        fileStream = null;
    }
    finally
    {
        if (fileStream != null)
```

```
            fileStream.Dispose();
        }
    }
```

该代码产生下列输出。

```
The streamWriter2.Close call generated an exception.
```

尽管第二个 StreamWriter 对象 streamWriter2 将内容写入一个被锁定的文件，但仅当执行 streamWriter2.Close 方法时，才会引发 IOException。

如果将本范例代码改写如下：

```
public static async Task CreateLockedFileWithUnlockAsync(string fileName)
{
    FileStream fileStream = null;
    try
    {
        fileStream = new FileStream(fileName,
                                    FileMode.Create,
                                    FileAccess.ReadWrite,
                                    FileShare.ReadWrite, 4096, useAsync: true);
        using (StreamWriter streamWriter = new StreamWriter(fileStream))
        {
            await streamWriter.WriteLineAsync("The First Line");
            await streamWriter.WriteLineAsync("The Second Line");
            await streamWriter.FlushAsync();

            // 锁定整个文件
            fileStream.Lock(0, fileStream.Length);

            // 尝试访问锁定的文件
            FileStream writeFileStream = null;
            try
            {
                writeFileStream = new FileStream(fileName,
                                            FileMode.Open,
                                            FileAccess.Write,
                                            FileShare.ReadWrite, 4096,
                                            useAsync: true);
                using (StreamWriter streamWriter2 =
                    new StreamWriter(writeFileStream))
                {
                    await streamWriter2.WriteAsync("foo");
                    fileStream.Unlock(0, fileStream.Length);
                    await streamWriter2.FlushAsync();
                }
                writeFileStream = null;
            }
            finally
            {
                if (writeFileStream != null)
                    writeFileStream.Dispose();
            }
        }
```

```
            fileStream = null;
        }
        finally
        {
            if (fileStream != null)
                fileStream.Dispose();
        }
    }
```

则不会引发任何异常。这是因为该代码解锁了最初锁定整个文件的 FileStream 对象。这一行为同时释放了 FileStream 对象所持有文件上所有的锁。在示例中，streamWriter2.Write("Foo") 方法已经将 Foo 写入流的缓冲区，但尚未刷新缓冲区，因此字符串 Foo 仍然等待刷新并写入实际的文件。当交错使用流的打开、锁定和关闭时，一定要牢记这种情况。代码中的错误有时在代码审查、单元测试和正式的 QA 测试中没有马上被发现，这会导致某些很难跟踪到的错误，因此在使用文件锁定时要小心对待。

8.5.4　参考

MSDN 文档中的“StreamWriter 类”“FileSystem 类”和“使用 Async 和 Await 的异步编程”主题。

8.6　等待文件系统中的动作发生

8.6.1　问题

你需要在文件系统中发生一个特定事件时得到通知，例如重命名一个文件或目录，文件大小增加或减少，删除一个文件或目录，创建一个文件或目录，甚至修改一个文件或目录的属性。但是，这一通知必须同步发生。换而言之，应用程序在指定行为出现在文件或目录中之前不能继续运行。

8.6.2　解决方案

可调用 FileSystemWatcher 类的 WaitForChanged 方法，以便同步等待一个事件通知。例 8-2 所示的 WaitForZipCreation 方法举例说明了这一点，它在继续执行下一行 WriteLine 语句代码之前等待一个行为被执行，更确切地说，是在 C:\ 驱动器上创建 Backup.zip 文件的行为。最后，我们分出一个任务来执行创建文件的工作。通过以 Task 来执行创建文件操作，我们允许当一个独立线程可用时在此线程上进行处理，并且 FileSystemWatcher 可以检测到文件创建。

例 8-2：WaitForZipCreation 方法

```
public static void WaitForZipCreation(string path, string fileName)
{
    if (string.IsNullOrWhiteSpace(path))
        throw new ArgumentNullException(nameof(path));
    if (string.IsNullOrWhiteSpace(fileName))
        throw new ArgumentNullException(nameof(fileName));
```

```
    FileSystemWatcher fsw = null;
    try
    {
        fsw = new FileSystemWatcher();
        string [] data = new string[] {path,fileName};
        fsw.Path = path;
        fsw.Filter = fileName;
        fsw.NotifyFilter = NotifyFilters.LastAccess | NotifyFilters.LastWrite
            | NotifyFilters.FileName | NotifyFilters.DirectoryName;

        // 运行此代码以生成监控的文件
        // 通常并不需要这样做,因为另一个源正在创建文件
        Task work = Task.Run(() =>
        {
            try
            {
                // 等待1秒
                Thread.Sleep(1000);
                // 在临时目录中创建文件
                if (data.Length == 2)
                {
                    string dataPath = data[0];
                    string dataFile = path + data[1];
                    Console.WriteLine($"Creating {dataFile} in task...");
                    FileStream fileStream = File.Create(dataFile);
                    fileStream.Close();
                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e.ToString());
            }
        });

        // 不必等待work任务完成
        // 因为我们通过FileSystemWatcher检测文件创建
        WaitForChangedResult result =
            fsw.WaitForChanged(WatcherChangeTypes.Created);
        Console.WriteLine($"{result.Name} created at {path}.");
    }
    catch(Exception e)
    {
        Console.WriteLine(e.ToString());
    }
    finally
    {
        // 清理
        File.Delete(fileName);
        fsw?.Dispose();
    }
}
```

8.6.3 讨论

WaitForChanged 方法返回一个包含表 8-1 中所列属性的 WaitForChangedResult 结构体。

表8-1：WaitForChangedResult属性

属　　性	描　　述
ChangeType	列出发生的改变类型。这种改变作为一个 WatcherChangeTypes 枚举被返回。该枚举的值可被组合求或
Name	持有发生改变的文件或目录名。如果文件或目录被重命名，那么该属性返回被修改后的名称。如果方法调用超时，则会将其值设置为 null
OldName	被改变的文件或目录的初始名。如果该文件或目录未被重命名，那么该属性将返回与 Name 属性相同的值。如果方法调用超时，则会将其值设置为 null
TimedOut	持有表示 WaitForChanged 方法超时的布尔值，超时则为 true，未超时则为 false

当前我们对 WaitForChanged 的调用方式可能会导致永远阻塞。为了防止永久挂起 WaitForChanged 调用，可以指定超时值为 3 秒，如下所示。

```
WaitForChangedResult result =
        fsw.WaitForChanged(WatcherChangeTypes.Created, 3000);
```

NotifyFilters 枚举允许指定等待的文件或文件夹的类型，如表 8-2 所示。

表8-2：NotifyFilters枚举

枚举值	定　　义
FileName	文件名
DirectoryName	目录名
Attributes	文件或文件夹属性
Size	文件或文件夹大小
LastWrite	文件或文件夹被写入内容的最后日期
LastAccess	文件或文件夹最后被访问的日期
CreationTime	文件或文件夹创建的时间
Security	文件或文件夹的安全设置

8.6.4 参考

MSDN 文档中的“FileSystemWatcher 类”“NotifyFilters 枚举”和“WaitForChanged-Result 结构”主题。

8.7 比较两个可执行模块的版本信息

8.7.1 问题

你需要编程比较两个可执行模块的版本信息。一个可执行模块是一个包含可执行代码的文件，如 .exe 或 .dll 文件。比较两个可执行模块版本信息的能力对以下几种情景中的应用程

序非常有用。

- 试图确定它是否具有要执行的所有“正确”片段。
- 确定一个通过反射动态加载的程序集。
- 查找散布在本地文件系统或网络的众多文件中某个文件或 .dll 的最新版本。

8.7.2 解决方案

使用 `CompareFileVersions` 方法比较可执行模块的版本信息。该方法接受两个文件名作为参数，包括它们的路径。每个模块的版本信息被获取并进行比较。该方法返回一个 `FileComparison` 枚举，其定义如下所示。

```
public enum FileComparison
{
    Error = 0,
    Newer = 1,
    Older = 2,
    Same = 3
}
```

`CompareFileVersions` 方法的代码如例 8-3 所示。

例 8-3：`CompareFileVersions` 方法

```
private static FileComparison ComparePart(int p1, int p2) =>
    p1 > p2 ? FileComparison.Newer :
        (p1 < p2 ? FileComparison.Older : FileComparison.Same);

public static FileComparison CompareFileVersions(string file1, string file2)
{
    if (string.IsNullOrWhiteSpace(file1))
        throw new ArgumentNullException(nameof(file1));
    if (string.IsNullOrWhiteSpace(file2))
        throw new ArgumentNullException(nameof(file2));

    FileComparison retValue = FileComparison.Error;
    // 获取版本信息
    FileVersionInfo file1Version = FileVersionInfo.GetVersionInfo(file1);
    FileVersionInfo file2Version = FileVersionInfo.GetVersionInfo(file2);

    retValue = ComparePart(file1Version.FileMajorPart,
        file2Version.FileMajorPart);
    if (retValue != FileComparison.Same)
    {
        retValue = ComparePart(file1Version.FileMinorPart, file2Version.FileMinorPart);
        if (retValue != FileComparison.Same)
        {
            retValue = ComparePart(file1Version.FileBuildPart,
                               file2Version.FileBuildPart);
            if (retValue != FileComparison.Same)
                retValue = ComparePart(file1Version.FilePrivatePart,
                        file2Version.FilePrivatePart);
        }
    }
```

```
        return retValue;
    }
```

8.7.3 讨论

并非所有的可执行模块都有版本信息。如果使用 `FileVersionInfo` 类加载了一个没有版本信息的模块，并不会引起异常，返回的对象引用也不会被设为 `null`。相反，你会得到一个合法的 `FileVersionInfo` 对象，其数据成员处于初始状态，对于 .NET 对象来说也就是 `null`。

程序集实际上有两个版本信息集，程序集清单中的版本信息和 PE（可移植可执行文件）文件版本信息。`FileVersionInfo` 读取程序集清单中的版本信息。

该方法的第一个动作是确定传递入 `file1` 和 `file2` 参数的文件是否实际存在。如果存在，调用 `FileVersionInfo` 类的 `GetVersionInfo` 静态方法去得到两个文件的版本信息。

`CompareFileVersions` 方法通过使用由 `GetVersionInfo` 返回的 `FileVersionInfo` 对象的以下几个属性，试图比较文件版本号的每个部分。

- `FileMajorPart`

  版本号的前 2 字节

- `FileMinorPart`

  版本号的第 3、4 字节

- `FileBuildPart`

  版本号的第 5、6 字节

- `FilePrivatePart`

  版本号的最后 2 字节

完整的版本号由四部分组成，构成一个代表文件版本号的 8 字节数字。

`CompareFileVersions` 方法首先比较两个文件的 `FileMajorPart` 版本信息。如果两者相等，那么比较两个文件的 `FileMinorPart` 版本信息。这一过程继续比较 `FileBuildPart`，最后比较 `FilePrivatePart` 版本信息值。如果所有四个部分都相等，那么文件被认为是具有相同的版本号。如果发现一个文件有着比另一个文件更高的版本号，那么就认为它是最近的版本。

8.7.4 参考

MSDN 文档中的“`FileVersionInfo` 类”主题。

8.8 查询系统上所有驱动器的信息

8.8.1 问题

你的应用程序需要知道一个驱动器（HDD、CD 驱动器、DVD 驱动器、蓝光驱动器等）是否可用并且是否写入或读出就绪，以及驱动器上是否拥有足够的可用空闲空间。

8.8.2 解决方案

使用 `DriveInfo` 类的各种属性，如下所示。

```
public static void DisplayAllDriveInfo()
{
    DriveInfo[] drives = DriveInfo.GetDrives();
    Array.ForEach(drives, drive =>
    {
        if (drive.IsReady)
        {
            Console.WriteLine($"Drive {drive.Name} is ready.");
            Console.WriteLine($"AvailableFreeSpace: {drive.AvailableFreeSpace}");
            Console.WriteLine($"DriveFormat: {drive.DriveFormat}");
            Console.WriteLine($"DriveType: {drive.DriveType}");
            Console.WriteLine($"Name: {drive.Name}");
            Console.WriteLine("RootDirectory.FullName: " +
                $"{drive.RootDirectory.FullName}");
            Console.WriteLine($"TotalFreeSpace: {drive.TotalFreeSpace}");
            Console.WriteLine($"TotalSize: {drive.TotalSize}");
            Console.WriteLine($"VolumeLabel: {drive.VolumeLabel}");
        }
        else
        {
            Console.WriteLine($"Drive {drive.Name} is not ready.");
        }
        Console.WriteLine();
    });
}
```

每个系统可能会不同，因而结果有异，但该代码将显示类似下列格式的内容。

```
Drive C:\ is ready.
AvailableFreeSpace: 143210795008
DriveFormat: NTFS
DriveType: Fixed
Name: C:\
RootDirectory.FullName: C:\
TotalFreeSpace: 143210795008
TotalSize: 159989886976
VolumeLabel: Vol1

Drive D:\ is ready.
AvailableFreeSpace: 0
DriveFormat: UDF
```

```
DriveType: CDRom
Name: D:\
RootDirectory.FullName: D:\
TotalFreeSpace: 0
TotalSize: 3305965568
VolumeLabel: Vol2

Drive E:\ is ready.
AvailableFreeSpace: 4649025536
DriveFormat: UDF
DriveType: CDRom
Name: E:\
RootDirectory.FullName: E:\
TotalFreeSpace: 4649025536
TotalSize: 4691197952
VolumeLabel: Vol3

Drive F:\ is not ready
```

IsReady 和 AvailableFreeSpace 属性特别引人注意。IsReady 属性确定驱动器是否准备好被查询、写入或读出，但并不十分可靠，因为其状态可能很快就会发生变化。如果使用 IsReady，那么要确保解决驱动器尚未就绪这种情况。AvailableFreeSpace 属性按字节返回驱动器上的空闲空间。

8.8.3　讨论

.NET Framework 中的 DriveInfo 类允许简单地查询系统中某个特定驱动器或者所有驱动器上的信息。要查询单个驱动器的信息，可以使用如例 8-4 所示的代码。

例 8-4：获得特定驱动器的信息

```
DriveInfo drive = new DriveInfo("D");
if (drive.IsReady)
    Console.WriteLine($"The space available on the D:\\ drive: " +
                $"{drive.AvailableFreeSpace}");
else
    Console.WriteLine("Drive D:\\ is not ready.");
```

我们注意到，只有驱动器名被传递给 DriveInfo 构造函数。驱动器名可以是大写，也可以是小写，这无关紧要。在 8.8.2 节的代码中，你注意到的下一件事是 IsReady 属性在使用驱动器或查询其属性之前都被检测其值是否为 true。如果未对该属性进行 true 值的测试，并且由于某种原因驱动器未就绪（例如，那时该 CD 没有放进驱动器中），那么将会返回一个声明“The device is not ready”的 System.IO.IOException。本范例的解决方案中未使用 DriveInfo 的构造函数，而是使用 DriveInfo 类的静态方法 GetDrives 以返回一个 DriveInfo 对象数组。该数组中的每个 DriveInfo 对象对应于当前系统中的一个驱动器。

DriveInfo 类的 DriveType 属性返回一个 DriveType 枚举类型的枚举值。该枚举值确定当前 DriveInfo 对象代表的驱动器类型。表 8-3 列出了 DriveType 枚举的不同值。

表8-3：DriveType枚举值

枚举值	描　　述
CDRom	可以是一个 CD-ROM、CD 刻录器、DVD-ROM、DVD 或蓝光刻录驱动器
Fixed	一个固定驱动器，例如 HDD。要注意，USB HDD 被归于此类
Network	一个网络驱动器
NoRootDirectory	该驱动器上未发现根目录
Ram	一个 RAM 磁盘
Removable	一个可移除的存储设备
Unknown	某种这里未列出的驱动器类型

DriveInfo 类有两个非常相似的属性：AvailableFreeSpace 和 TotalFreeSpace。这两个属性在大多数情况下返回相同的值，但 AvailableFreeSpace 还考虑到某个特定驱动器的磁盘配额信息。在 Windows 资源管理器中可通过右击一个驱动器然后选择“属性”弹出菜单项来找到磁盘配额信息。这一操作会显示该驱动器的“属性”页面。在该“属性”页面中，单击“配额”选项卡可查看该驱动器的配额信息。如果“启用配额管理”复选框未选中，则磁盘配额管理被禁用，并且 AvailableFreeSpace 和 TotalFreeSpace 属性值应当是相等的。

8.8.4　参考

MSDN 文档中的“DriveInfo 类”主题。

8.9　压缩和解压缩文件

8.9.1　问题

你需要一种使用一个基于流的类来压缩文件的方法，并且不受 Framework 类施加的 4 GB 限制。此外还需要一种解压缩文件的方法，使得你能够读回文件内容。

8.9.2　解决方案

使用 System.IO.Compression.DeflateStream 或 System.IO.Compression.GZipStream 类，通过使用一个“数据块化”的例程来读取压缩过的数据并将其写入一个文件。例 8-5 显示的 CompressFileAsync、DecompressFileAsync 和 Decompress 方法举例说明了如何使用这些类即时地压缩和解压缩文件。

例 8-5：CompressFileAsync 和 DecompressFileAsync 方法

```
/// <summary>
/// 将源文件压缩到目标文件。
/// 通过使用1 MB的数据块来完成这一操作,从而不会溢出内存使用
/// </summary>
/// <param name="sourceFile">未压缩的文件</param>
/// <param name="destinationFile">压缩过的文件</param>
/// <param name="compressionType">要使用的压缩类型</param>
public static async Task CompressFileAsync(string sourceFile,
```

```
                                string destinationFile,
                                CompressionType compressionType)
{
    if (string.IsNullOrWhiteSpace(sourceFile))
        throw new ArgumentNullException(nameof(sourceFile));

    if (string.IsNullOrWhiteSpace(destinationFile))
        throw new ArgumentNullException(nameof(destinationFile));

    FileStream streamSource = null;
    FileStream streamDestination = null;
    Stream streamCompressed = null;

    int bufferSize = 4096;
    using (streamSource = new FileStream(sourceFile,
            FileMode.OpenOrCreate, FileAccess.Read, FileShare.None,
            bufferSize, useAsync: true))
    {
        using (streamDestination = new FileStream(destinationFile,
            FileMode.OpenOrCreate, FileAccess.Write, FileShare.None,
            bufferSize, useAsync: true))
        {
            // 读取1 MB的数据块并将其压缩
            long fileLength = streamSource.Length;

            // 写出fileLength的大小
            byte[] size = BitConverter.GetBytes(fileLength);
            await streamDestination.WriteAsync(size, 0, size.Length);

            long chunkSize = 1048576; // 1 MB
            while (fileLength > 0)
            {
                // 读取数据块
                byte[] data = new byte[chunkSize];
                await streamSource.ReadAsync(data, 0, data.Length);

                // 压缩数据块
                MemoryStream compressedDataStream =
                    new MemoryStream();

                if (compressionType == CompressionType.Deflate)
                    streamCompressed =
                        new DeflateStream(compressedDataStream,
                            CompressionMode.Compress);
                else
                    streamCompressed =
                        new GZipStream(compressedDataStream,
                            CompressionMode.Compress);

                using (streamCompressed)
                {
                    // 将数据块写入压缩数据流
                    await streamCompressed.WriteAsync(data, 0, data.Length);
                }
                // 获取压缩数据块的字节数
```

```
                byte[] compressedData =
                    compressedDataStream.GetBuffer();

                // 写出数据块大小
                size = BitConverter.GetBytes(chunkSize);
                await streamDestination.WriteAsync(size, 0, size.Length);

                // 写出压缩过的大小
                size = BitConverter.GetBytes(compressedData.Length);
                await streamDestination.WriteAsync(size, 0, size.Length);

                // 写出压缩数据块
                await streamDestination.WriteAsync(compressedData, 0,
                    compressedData.Length);

                // 从文件大小中减去数据块大小
                fileLength -= chunkSize;

                // 如果剩余文件大小小于数据块大小,使用剩余文件大小
                if (fileLength < chunkSize)
                    chunkSize = fileLength;
            }
        }
    }
}

/// <summary>
/// 此函数将解压通过CompressFileAsync函数创建的块压缩的文件
/// </summary>
/// <param name="sourceFile">压缩文件</param>
/// <param name="destinationFile">目标文件</param>
/// <param name="compressionType">要使用的压缩类型</param>
public static async Task DecompressFileAsync(string sourceFile,
                                string destinationFile,
                                CompressionType compressionType)
{
    if (string.IsNullOrWhiteSpace(sourceFile))
        throw new ArgumentNullException(nameof(sourceFile));
    if (string.IsNullOrWhiteSpace(destinationFile))
        throw new ArgumentNullException(nameof(destinationFile));

    FileStream streamSource = null;
    FileStream streamDestination = null;
    Stream streamUncompressed = null;

    int bufferSize = 4096;
    using (streamSource = new FileStream(sourceFile,
            FileMode.OpenOrCreate, FileAccess.Read, FileShare.None,
            bufferSize, useAsync: true))
    {
        using (streamDestination = new FileStream(destinationFile,
            FileMode.OpenOrCreate, FileAccess.Write, FileShare.None,
            bufferSize, useAsync: true))
        {
            // 读取fileLength大小
```

```
// 读取块大小
byte[] size = new byte[sizeof(long)];
await streamSource.ReadAsync(size, 0, size.Length);
// 将size转换回数值
long fileLength = BitConverter.ToInt64(size, 0);
long chunkSize = 0;
int storedSize = 0;
long workingSet = Process.GetCurrentProcess().WorkingSet64;
while (fileLength > 0)
{
    // 读取块大小
    size = new byte[sizeof(long)];
    await streamSource.ReadAsync(size, 0, size.Length);
    // 将size转换回数值
    chunkSize = BitConverter.ToInt64(size, 0);
    if (chunkSize > fileLength ||
        chunkSize > workingSet)
        throw new InvalidDataException();

    // 读取压缩过的块大小
    size = new byte[sizeof(int)];
    await streamSource.ReadAsync(size, 0, size.Length);
    // 将size转换回数值
    storedSize = BitConverter.ToInt32(size, 0);
    if (storedSize > fileLength ||
        storedSize > workingSet)
        throw new InvalidDataException();

    if (storedSize > chunkSize)
        throw new InvalidDataException();

    byte[] uncompressedData = new byte[chunkSize];
    byte[] compressedData = new byte[storedSize];
    await streamSource.ReadAsync(compressedData, 0,
        compressedData.Length);

    // 解压缩数据块
    MemoryStream uncompressedDataStream =
        new MemoryStream(compressedData);

    if (compressionType == CompressionType.Deflate)
        streamUncompressed =
            new DeflateStream(uncompressedDataStream,
                CompressionMode.Decompress);
    else
        streamUncompressed =
            new GZipStream(uncompressedDataStream,
                CompressionMode.Decompress);

    using (streamUncompressed)
    {
        // 读取压缩数据流中的数据块
        await streamUncompressed.ReadAsync(uncompressedData, 0,
            uncompressedData.Length);
    }
```

```
                // 写出未压缩过的数据块
                await streamDestination.WriteAsync(uncompressedData, 0,
                    uncompressedData.Length);

                // 从文件大小中减去数据块大小
                fileLength -= chunkSize;

                // 如果剩余文件大小小于数据块大小,使用剩余文件大小
                if (fileLength < chunkSize)
                    chunkSize = fileLength;
            }
        }
    }
```

CompressionType 枚举定义如下所示。

```
public enum CompressionType
{
    Deflate,
    GZip
}
```

8.9.3 讨论

CompressFileAsync 方法接受一个指向要压缩的源文件的路径、一个指向被压缩文件目的地的路径以及一个指示使用哪种压缩算法（Deflate 或 GZip）的 CompressionType 枚举值。该方法生成一个包含了压缩数据的文件。

DecompressFileAsync 方法接受一个指向要解压缩的源压缩文件的路径，一个指向解压缩文件目的地的路径以及一个指示使用哪种解压缩算法（Deflate 或 GZip）的 CompressionType 枚举值。

例 8-6 所示的 TestCompressNewFileAsync 方法运用了 8.9.2 节中定义的 CompressFileAsync 和 DecompressFileAsync 方法。

例 8-6：使用 CompressFileAsync 和 DecompressFileAsync 方法

```
public static async void TestCompressNewFileAsync()
{
    byte[] data = new byte[10000000];
    for (int i = 0; i < 10000000; i++)
        data[i] = (byte)i;


    using(FileStream fs =
        new FileStream(@"C:\NewNormalFile.txt",
            FileMode.OpenOrCreate, FileAccess.ReadWrite, FileShare.None,
            4096, useAsync:true))
    {
        await fs.WriteAsync(data, 0, data.Length);
    }
```

```
        await CompressFileAsync(@"C:\NewNormalFile.txt", @"C:\NewCompressedFile.txt",
            CompressionType.Deflate);

        await DecompressFileAsync(@"C:\NewCompressedFile.txt",
            @"C:\NewDecompressedFile.txt",
            CompressionType.Deflate);

        await CompressFileAsync(@"C:\NewNormalFile.txt", @"C:\NewGZCompressedFile.txt",
            CompressionType.GZip);

        await DecompressFileAsync(@"C:\NewGZCompressedFile.txt",
            @"C:\NewGZDecompressedFile.txt",
            CompressionType.GZip);

        // 正常文件大小 == 10 000 000字节
        // GZip压缩文件大小 == 84 362
        // Deflate压缩文件大小 == 42 145
        // .NET 4.5之前版本GZip压缩文件大小 == 155 204
        // .NET 4.5之前版本Deflate压缩文件大小 == 155 168
        // 36字节数据与GZip算法的CRC有关
    }
```

当此测试代码运行时，我们将得到三个大小不一的文件。第一个文件 NewNormalFile.txt 的大小是 10 000 000 字节。NewCompressedFile.txt 文件为 42 145 字节，最后一个文件 NewGzCompressedFile.txt 为 84 362 字节。如你所见，使用 `DeflateStream` 类和 `GZipStream` 类压缩的文件大小之间并没有太大差异，其原因是这两种压缩类使用相同的压缩 / 解压缩算法（即 RFC 1951: Deflate 1.3 规范中描述的无损 Deflate 算法）。

在 .NET 4.5 中更新过的 `GZipStream` 和 `DeflateStream` 类使用 `zlib library`（http://www.zlib.net/）来执行压缩，这个库提高了压缩比。如果在之前版本的 .NET Framework 上运行例 8-7 所示的旧版本 `CompressFile` 和 `DecompressFile` 方法，你可以看到这一对比。

例 8-7：用于 .NET 4.5 之前版本的 `CompressFile` 和 `DecompressFile` 方法

```
    /// <summary>
    /// 将源文件压缩到目标文件。
    /// 通过使用1 MB的数据块来完成这一操作,从而不会溢出内存使用
    /// </summary>
    /// <param name="sourceFile">未压缩的文件</param>
    /// <param name="destinationFile">压缩过的文件</param>
    /// <param name="compressionType">要使用的压缩类型</param>
    public static void CompressFile(string sourceFile,
                                    string destinationFile,
                                    CompressionType compressionType)
    {
        if (sourceFile != null)
        {
            FileStream streamSource = null;
            FileStream streamDestination = null;
            Stream streamCompressed = null;

            using (streamSource = File.OpenRead(sourceFile))
            {
                using (streamDestination = File.OpenWrite(destinationFile))
```

```
{
    // 读取1 MB的数据块并将其压缩
    long fileLength = streamSource.Length;

    // 写出fileLength的大小
    byte[] size = BitConverter.GetBytes(fileLength);
    streamDestination.Write(size, 0, size.Length);

    long chunkSize = 1048576; // 1 MB
    while (fileLength > 0)
    {
        // 读取数据块
        byte[] data = new byte[chunkSize];
        streamSource.Read(data, 0, data.Length);

        // 压缩数据块
        MemoryStream compressedDataStream =
            new MemoryStream();

        if (compressionType == CompressionType.Deflate)
            streamCompressed =
                new DeflateStream(compressedDataStream,
                    CompressionMode.Compress);
        else
            streamCompressed =
                new GZipStream(compressedDataStream,
                    CompressionMode.Compress);

        using (streamCompressed)
        {
            // 将数据块写入压缩数据流
            streamCompressed.Write(data, 0, data.Length);
        }
        // 获取压缩数据块的字节数
        byte[] compressedData =
            compressedDataStream.GetBuffer();

        // 写出数据块大小
        size = BitConverter.GetBytes(chunkSize);
        streamDestination.Write(size, 0, size.Length);

        // 写出压缩过的大小
        size = BitConverter.GetBytes(compressedData.Length);
        streamDestination.Write(size, 0, size.Length);

        // 写出压缩数据块
        streamDestination.Write(compressedData, 0,
            compressedData.Length);

        // 从文件大小中减去数据块大小
        fileLength -= chunkSize;

        // 如果剩余文件大小小于数据块大小,使用剩余文件大小
        if (fileLength < chunkSize)
            chunkSize = fileLength;
```

```
                }
            }
        }
    }
}

/// <summary>
/// 此函数将解压通过CompressFileAsync函数创建的块压缩的文件
/// </summary>
/// <param name="sourceFile">压缩文件</param>
/// <param name="destinationFile">目标文件</param>
/// <param name="compressionType">要使用的压缩类型</param>
public static void DecompressFile(string sourceFile,
                                string destinationFile,
                                CompressionType compressionType)
{
    FileStream streamSource = null;
    FileStream streamDestination = null;
    Stream streamUncompressed = null;

    using (streamSource = File.OpenRead(sourceFile))
    {
        using (streamDestination = File.OpenWrite(destinationFile))
        {
            // 读取fileLength大小
            // 读取块大小
            byte[] size = new byte[sizeof(long)];
            streamSource.Read(size, 0, size.Length);
            // 将size转换回数值
            long fileLength = BitConverter.ToInt64(size, 0);
            long chunkSize = 0;
            int storedSize = 0;
            long workingSet = Process.GetCurrentProcess().WorkingSet64;
            while (fileLength > 0)
            {
                // 读取块大小
                size = new byte[sizeof(long)];
                streamSource.Read(size, 0, size.Length);
                // 将size转换回数值
                chunkSize = BitConverter.ToInt64(size, 0);
                if (chunkSize > fileLength ||
                    chunkSize > workingSet)
                    throw new InvalidDataException();

                // 读取压缩过的块大小
                size = new byte[sizeof(int)];
                streamSource.Read(size, 0, size.Length);
                // 将size转换回数值
                storedSize = BitConverter.ToInt32(size, 0);
                if (storedSize > fileLength ||
                    storedSize > workingSet)
                    throw new InvalidDataException();

                if (storedSize > chunkSize)
                    throw new InvalidDataException();
```

```
                byte[] uncompressedData = new byte[chunkSize];
                byte[] compressedData = new byte[storedSize];
                streamSource.Read(compressedData, 0,
                    compressedData.Length);

                // 解压缩数据块
                MemoryStream uncompressedDataStream =
                    new MemoryStream(compressedData);

                if (compressionType == CompressionType.Deflate)
                    streamUncompressed =
                        new DeflateStream(uncompressedDataStream,
                            CompressionMode.Decompress);
                else
                    streamUncompressed =
                        new GZipStream(uncompressedDataStream,
                            CompressionMode.Decompress);

                using (streamUncompressed)
                {
                    // 读取压缩数据流中的数据块
                    streamUncompressed.Read(uncompressedData, 0,
                        uncompressedData.Length);
                }

                // 写出未压缩过的数据块
                streamDestination.Write(uncompressedData, 0,
                    uncompressedData.Length);

                // 从文件大小中减去数据块大小
                fileLength -= chunkSize;

                // 如果剩余文件大小小于数据块大小,使用剩余文件大小
                if (fileLength < chunkSize)
                    chunkSize = fileLength;
            }
        }
    }
}
```

你可能会感到奇怪：如果它们使用相同的算法，那为什么选择某一个类而不是另一个类？一个非常好的理由是，`GZipStream` 类为压缩过的数据添加了一个 CRC（循环冗余校验），以确定它是否损坏。如果数据损坏，则会引发一个 `InvalidDataException` 异常，带有 The CRC in GZip footer does not match the CRC calculated from the decompressed data 的表达。通过捕获这一异常，就可以确定你的数据是否已损坏。

在 `Decompess` 方法中，可能会引发某些 `InvalidDataException` 实例。

```
// 读取块大小
size = new byte[sizeof(long)];
streamSource.Read(size, 0, size.Length);
// 将size转换回数值
chunkSize = BitConverter.ToInt64(size, 0);
```

```
if (chunkSize > fileLength || chunkSize > workingSet)
    throw new InvalidDataException();

// 读取压缩过的块大小
size = new byte[sizeof(int)];
streamSource.Read(size, 0, size.Length);
// 将size转换回数值
storedSize = BitConverter.ToInt32(size, 0);
if (storedSize > fileLength || storedSize > workingSet)
    throw new InvalidDataException();
if (storedSize > chunkSize)
    throw new InvalidDataException();

byte[] uncompressedData = new byte[chunkSize];
byte[] compressedData = new byte[storedSize];
```

该代码读取可能已经被篡改的缓冲区中的数据，因此需要进行检查，这不仅是出于稳定性，也是出于安全方面的考虑。因为 `Decompress` 将根据从缓冲区中读取的数值进行实际的内存分配，所以需要小心检查这些数值，而且我们不想在无意中引入已被注入到流中的其他代码。此处进行的基本检查是为了确保以下事项。

- 数据块的大小不大于文件长度。
- 数据块的大小不大于当前程序的工作集。
- 压缩的数据块大小不大于文件长度。
- 压缩的数据块大小不大当于前程序的工作集。
- 压缩的数据块大小不大于实际的数据块大小。

8.9.4 参考

MSDN 文档中的“`DeflateStream` 类”和“`GZipStream` 类”主题。

第 9 章

网络和Web

9.0 简介

连接性在解决方案中变得比以往更加重要，.NET Framework 提供了多种方法来帮助支持这一需求。.NET 提供了许多较低级别的类，使其网络编程比之前的诸多环境更加容易。有大量的功能可帮助你完成以下几项工作。

- 建立网络感知的应用程序
- 通过 FTP 下载文件
- 发送和接收 HTTP 请求
- 直接使用 TCP/IP 和套接字获得更高程度的控制

在 Microsoft 尚未提供托管类以访问联网功能的领域中（例如 `WinInet` API 所暴露的用于互联网连接设置的某些方法），总是可以使用 P/Invoke，因此可以使用 Win32 API 编码；在本章中将会探讨此方法。使用 `System.Net` 命名空间中可用的所有功能，可以快速地编写网络应用程序。

除了较低级别的网络支持，.NET 也积接迎合万维网的发展，已经对大多数 .NET 开发人员目前在构建其解决方案时遇到的各种问题提供了 Web 支持。Web 服务（基于 REST 和基于 SOAP）被广泛应用，ASP.NET 在 Web 应用领域是主角之一。由于对处理 HTML 和 TCP/IP 名称解析的一般需求，以及统一资源指示器（URI）和统一资源定位器（URL）应用越来越广泛，开发人员需要工具来帮助自己全神贯注于构建他们能够提供的最好的 Web 交互应用程序。本章关注涉及 Web 时出现的某些编程边缘问题。本章并非 Web 服务或 ASP.NET 的教程，而是涵盖了开发人员可以在 ASP.NET 应用或服务中使用，以及在其他基于 C# 且与网络和 Web 交互的应用程序中使用的一些功能。

9.1 处理Web服务器错误

9.1.1 问题

你获得了一个来自 Web 服务器的响应，并且希望确保处理初始请求时没有发生错误，例如连接失败、被重定向、超时或证书验证失败。你希望避免检查所有不同的可用响应代码。

9.1.2 解决方案

检查 `HttpWebResponse` 类的 `StatusCode` 属性以确定这个 `StatusCode` 属于哪个类别的状态，并返回一个代码类别的枚举值（`ResponseCategories`）。这一技术允许使用一种更具普遍性的方式来处理响应代码。

```
public static ResponseCategories CategorizeResponse(HttpWebResponse httpResponse)
{
    // 为了处理未来在HttpStatusCode中定义的更多成功代码,
    // 此处我们将检查"成功"代码范围,而不是使用HttpStatusCode枚举,
    // 因为它重载了某些值
    int statusCode = (int)httpResponse.StatusCode;
    if ((statusCode >= 100) && (statusCode <= 199))
    {
        return ResponseCategories.Informational;
    }
    else if ((statusCode >= 200) && (statusCode <= 299))
    {
        return ResponseCategories.Success;
    }
    else if ((statusCode >= 300) && (statusCode <= 399))
    {
        return ResponseCategories.Redirected;
    }
    else if ((statusCode >= 400) && (statusCode <= 499))
    {
        return ResponseCategories.ClientError;
    }
    else if ((statusCode >= 500) && (statusCode <= 599))
    {
        return ResponseCategories.ServerError;
    }
    return ResponseCategories.Unknown;
}
```

`ResponseCategories` 枚举的定义如下所示。

```
public enum ResponseCategories
{
    Unknown,        // 未知代码(<100或>599)
    Informational,  // 消息代码(100<=199)
    Success,        // 成功代码(200<=299)
    Redirected,     // 重定向代码(300<=399)
    ClientError,    // 客户端错误代码(400<=499)
```

```
    ServerError     // 服务器错误代码(500<=599)
}
```

9.1.3 讨论

在 HTTP 响应中有着五种不同的状态类别，如表 9-1 所示。

表9-1：HTTP响应状态码的类别

类　别	可用的范围	定义的HttpStatusCode范围
消息	100~199	100~101
成功	200~299	200~206
重定向	300~399	300~307
客户端错误	400~499	400-426
服务器错误	500~599	500~505

Microsoft 在 .NET Framework 中定义的每个状态码皆被赋予一个 `HttpStatusCode` 枚举中的枚举值。这些状态码反映了当提交一个请求时可能发生的情况。Web 服务器可以自由返回位于可用范围内的一个状态码，即使是大多数的商业 Web 服务器尚未定义的值。针对 HTTP/1.1 的 RFC 2616 中的第 10 节列出了已定义的状态码。

你想要指出请求的状态所属的概括类别。通过检查 `HttpResponse.StatusCode` 属性，将它与已定义的用于 HTTP 的状态码范围进行比较，返回适当的 `ResponseCategories` 值可以实现操作。

当处理 `HttpStatusCode` 时，你会注意到存在某些映射到相同状态码值的特定 `HttpStatusCode` 标志。一个例子是 `HttpStatusCode.Ambiguous` 和 `HttpStatusCode.MultipleChoices`，它们都映射到了 HTTP 状态码 300。如果试图在一个关于 `HttpStatusCode` 的 `switch` 语句中使用这两者，将会得到下列错误，因为 C# 编译器无法发现其差异。

```
error CS0152: The label 'case 300:' already occurs in this switch statement.
```

9.1.4 参考

《HTTP 权威指南》，英文版由 O’Reilly 出版；MSDN 文档中的“`HttpStatusCode` 枚举”主题；HTTP/1.1 RFC 2616 第 10 节“状态码”（http://t.cn/GPNNm）。

9.2 与Web服务器通信

9.2.1 问题

你希望用 `GET` 或 `POST` 请求的形式向一个 Web 服务器发送一个请求。向 Web 服务器发送完请求后，你希望得到来自 Web 服务器的请求结果（响应）。

9.2.2 解决方案

结合使用 HttpWebRequest 类与 WebRequest 类，创建一个请求并发送至一个服务器。

获取资源的 Uri（统一资源标识符；在 RFC 3986 中定义），请求（GET 或 POST）中使用的方法，以及要发送的数据（仅针对 POST 请求），并使用这些信息创建一个 HttpWebRequest，如例 9-1 所示。

例 9-1：与 Web 服务器通信

```
using System.Net;
using System.IO;
using System.Text;

// GET重载
public static HttpWebRequest GenerateHttpWebRequest(Uri uri)
{
    // 创建初始请求
    HttpWebRequest httpRequest = (HttpWebRequest)WebRequest.Create(uri);
    // 返回请求
    return httpRequest;
}

// POST重载
public static HttpWebRequest GenerateHttpWebRequest(Uri uri,
    string postData,
    string contentType)
{
    // 创建初始请求
    HttpWebRequest httpRequest = GenerateHttpWebRequest(uri);

    // 获得请求的字节数组,需要预先进行转义
    byte[] bytes = Encoding.UTF8.GetBytes(postData);

    // 设置要提交数据的内容类型
    httpRequest.ContentType = contentType;
        //"application/x-www-form-urlencoded"; 对于表单
         //"application/json" 对于json数据
         //"application/xml" 对于xml数据

    // 设置要提交字符串的内容长度
    httpRequest.ContentLength = postData.Length;

    // 获取请求流,并写入发布的数据
    using (Stream requestStream = httpRequest.GetRequestStream())
    {
        requestStream.Write(bytes, 0, bytes.Length);
    }
    // 返回请求
    return httpRequest;
}
```

一旦你拥有了一个 HttpWebRequest，就可以发送请求并使用 GetResponse 方法获得响应。它将新近生成的 HttpWebRequest 作为输入，返回一个 HttpWebResponse。下列示例针对

http://localhost/mysite 网站的 index.aspx 页面执行一个 GET。

```
HttpWebRequest request =
    GenerateHttpWebRequest(new Uri("http://localhost/mysite/index.aspx"));

using(HttpWebResponse response = (HttpWebResponse) request.GetResponse())
{
    // 下一行代码使用了范例9.1(即9.1节)中的CategorizeResponse
    if(CategorizeResponse(response)==ResponseCategories.Success)
    {
        Console.WriteLine("Request succeeded");
    }
}
```

你生成 HttpWebRequest，发送它并得到 HttpWebResponse，然后使用范例 9.1（即 9.1 节）中的 CategorizeResponse 方法以检查是否成功。

9.2.3 讨论

WebRequest 和 WebResponse 类封装了执行基本 Web 通信的所有功能。HttpWebRequest 和 HttpWebResponse 是从这些类派生而来的，并且提供了特定的 HTTP 支持。

在最基础级别上，要执行一个基于 HTTP 的 Web 事务，可以使用 WebRequest 类上的 Create 方法获得一个可类型转换为 HttpWebRequest 的 WebRequest（只要模式是 http:// 或 https:// 即可）。然后在调用 GetResponse 方法时，这个 HttpWebRequest 被提交到相应的 Web 服务器，并且返回一个之后可用于检查响应数据的 HttpWebResponse。

9.2.4 参考

MSDN 文档中的“WebRequest 类”“WebResponse 类”“HttpWebRequest 类”和“HttpWebResponse 类”主题，以及统一资源标识符 RFC（http://t.cn/R53QltB）。

9.3 通过代理服务器

9.3.1 问题

许多公司都有一个代理服务器［有时也称为 Web 代理（web proxy)］，允许雇员访问互联网，同时防止外部人员访问公司的内部网络。问题在于要创建一个从公司内部访问互联网的一个应用程序，就必须首先连接到代理服务器，然后通过它发送信息，而不是直接连接到一个互联网 Web 服务器。

9.3.2 解决方案

要通过某个特定的代理服务器成功获取一个 HttpWebRequest，你需要使用一些设置来设置一个 WebProxy 对象，这些设置用于验证对某个给定代理的特定请求。因为这一功能对于所有请求都是通用的，所以你可以创建 AddProxyInfoToRequest 方法。

```
public static HttpWebRequest AddProxyInfoToRequest(HttpWebRequest httpRequest,
    Uri proxyUri,
    string proxyId,
    string proxyPassword,
    string proxyDomain)
{
    if (httpRequest == null)
        throw new ArgumentNullException(nameof(httpRequest));

    // 创建代理对象
    WebProxy proxyInfo = new WebProxy();
    // 添加要使用的代码服务的地址
    proxyInfo.Address = proxyUri;
    // 将其设置为针对本地地址跳过代理服务器
    proxyInfo.BypassProxyOnLocal = true;
    // 添加提供给代理服务器的任意凭据信息
    proxyInfo.Credentials = new NetworkCredential(proxyId,
        proxyPassword,
        proxyDomain);
    // 将代理信息赋予请求对象
    httpRequest.Proxy = proxyInfo;

    // 返回请求
    return httpRequest;
}
```

如果所有请求都要通过相同的代理服务器，在 Framework 的 1.x 版本中，使用 GlobalProxySelection 类上的静态 Select 方法为 WebRequest 建立代理服务器设置。在 1.x 版本之后，应当使用 WebRequest.DefaultWebProxy 属性，代码如下所示。

```
// 将所有请求都设置为通过此Uri指示的代理
Uri proxyURI = new Uri("http://webproxy:80");

// 在1.1中，你需要这样做：
//GlobalProxySelection.Select = new WebProxy(proxyURI);

// 现在，在2.0以及更高版本中，你需要这样做：
WebRequest.DefaultWebProxy = new WebProxy(proxyURI);
```

9.3.3 讨论

AddProxyInfoToRequest 获得代理服务器的 URI 并创建一个 Uri 对象，它用于构建 WebProxy 对象。WebProxy 对象被设置为对本地地址绕过代理服务器，然后使用凭据信息创建一个 NetworkCredential 对象。NetworkCredential 对象代表随后在代理服务器完成请求所必需的验证信息，并且被赋给了 WebProxy.Credentials 属性。一旦 WebProxy 对象完成，它便被赋予 HttpWebRequest 的 Proxy 属性，此时请求就准备好被提交了。

要从 Internet Explorer 中获得针对当前用户的代理服务器设置，可以使用 System.Net.WebRequest.GetSystemWebProxy 方法，然后将返回的 IWebProxy 赋予 HttpWebRequest 上的代理服务器或者 WebRequest 上的 DefaultWebProxy 属性，代码如下所示。

```
WebRequest.DefaultWebProxy = WebRequest.GetSystemWebProxy();
```

9.3.4 参考

MSDN 文档中的“WebProxy 类”“NetworkCredential 类”和“HttpWebRequest 类”主题。

9.4 从一个URL获取HTML

9.4.1 问题

你需要获得从 Web 服务器返回的 HTML 以便对感兴趣的项进行检查。例如，你可以从返回的 HTML 中检查到其他页面的链接或者新闻站点中的标题。

9.4.2 解决方案

你可以使用范例 9.1（即 9.1 节）和范例 9.2（即 9.2 节）中建立的用于 Web 通信的方法构造 HTTP 请求并验证响应，然后就可以通过 HttpWebResponse 对象的 ResponseStream 属性获得 HTML，代码如下所示。

```
public static async Task<string> GetHtmlFromUrlAsync(Uri url)
{
    string html = string.Empty;
    HttpWebRequest request = GenerateHttpWebRequest(url);
    using(HttpWebResponse response =
        (HttpWebResponse) await request.GetResponseAsync())
    {
        if (CategorizeResponse(response) == ResponseCategories.Success)
        {
            // 获得响应流
            Stream responseStream = response.GetResponseStream();
            // 使用一个理解UTF8的流读取器
            using(StreamReader reader =
                new StreamReader(responseStream, Encoding.UTF8))
            {
                html = reader.ReadToEnd();
            }
        }
    }
    return html;
}
```

9.4.3 讨论

GetHtmlFromUrlAsync 方法使用 GenerateHttpWebRequest 和 GetResponse 方法获得一个网页，使用 CategorizeResponse 方法验证响应。之后，一旦有了一个有效的响应，就开始查找返回的 HTML。

HttpWebResponse 上的 GetResponseStream 方法提供了对在 System.IO.Stream 对象中返回的消息体的访问。要读取数据，可使用响应流和 Encoding 类的 UTF8 属性来实例化一个 StreamReader，从而允许从流中正确地读取 UTF8 编码的文本数据。然后调用

StreamReader 的 ReadToEnd 方法，将所有内容放入命名为 html 的字符串中，并且返回它。

9.4.4　参考

MSDN 文档中的“HttpWebResponse.GetResponseStream 方法”“Stream 类”和“StringBuilder 类”主题。

9.5　使用Web浏览器控件

9.5.1　问题

你需要在基于 WinForms 的应用程序中显示基于 HTML 的内容。

9.5.2　解决方案

使用 System.Windows.Forms.WebBrowser 类将 Web 浏览器功能嵌入到应用程序中。图 9-1 所示的 Cheapo-Browser 展示了这一控件的某些功能。



图 9-1：Web 浏览器控件

虽然这不是一个达到发布产品质量的用户界面（所以被称为 Cheapo-Browser），但它可用于选择一个 Web 地址，显示内容，向前和向后导航，取消请求，返回主页，直接向控件添加 HTML，打印或保存 HTML，以及启用或者取消浏览器窗口内部的上下文菜单。WebBrowser 控件的能力不止于此，但本范例的意图在于让用户尝试一下可以做到什么。进一步探索其能力来获知它可能满足的其他需求是很值得的。

当你添加自己的 HTML（<h1>Hey you added some HTML!</h1>）时，它的显示如图 9-2 所示。



图 9-2：向 Cheapo-Browser 添加 HTML

完成这一任务的代码如下所示，它相当简单。

```
this._webBrowser.Document.Body.InnerHtml = "<h1>Hey you added some HTML!</h1>";
```

导航至一个 Web 页面的代码如下所示，它与上面的代码一样简单。

```
Uri uri = new Uri(this._txtAddress.Text);
this._webBrowser.Navigate(uri);
```

关于导航处理方式的一个美妙之处在于可以订阅 Navigated 事件，以便在导航完成时得到

通知。这允许代码在线程中进行导航并在它被完全加载后继续处理。该事件提供了一个 WebBrowserNavigatedEventArgs 类，它拥有一个 Uri 属性，告知被导航至文档的 URL，代码如下所示。

```
private void _webBrowser_Navigated(object sender, WebBrowserNavigatedEventArgs e)
{
    // 更新最终到达的url,以处理从原始Uri重定向的情况
    this._txtAddress.Text = e.Url.ToString();
    this._btnBack.Enabled = this._webBrowser.CanGoBack;
    this._btnForward.Enabled = this._webBrowser.CanGoForward;
}
```

9.5.3　讨论

在 .NET Framework 的 1.x 版本中，在 WinForms 应用程序中内嵌一个 Web 浏览器更加困难并且容易出错。现在有了一种基于 .NET 的 Web 浏览器控件来处理这些难题。当你试图与浏览器事件挂接时，不再需要费力处理可能出现的 COM interop 问题了。这是一个好机会，可以使桌面应用程序与 Web 应用程序的界限更加模糊，灵活地结合使用 Web 和富客户端的功能。

9.5.4　参考

MSDN 文档中的“WebBrowser 类”主题。

9.6　以编程方式预构建一个ASP.NET网站

9.6.1　问题

你希望预构建自己的网站，以避免编译延迟和源代码需要存放在服务器上的托管场景。

9.6.2　解决方案

使用 ClientBuildManager 将自己的网站预构建为一个程序集。要预构建网站，必须指定以下各项。

- 用于 Web 应用程序的虚拟目录
- 指向 Web 应用程序目录的物理路径
- 你希望构建 Web 应用程序的位置
- 帮助控制编译的标志

要预构建本示例代码中的 Web 应用程序，首先要获得 Web 应用程序所在的目录，然后提供一个虚拟目录名和用于构建 Web 应用程序的位置，代码如下所示。

```
string cscbWebPath = GetWebAppPath();

if(cscbWebPath.Length > 0)
{
```

```
    string appVirtualDir = @"CSCBWeb";
    string appPhysicalSourceDir = cscbWebPath;

    // 将目标设置为邻近目录,因为其不能与源码在同一个目录树中,
    // 否则构建管理器将报错
    string appPhysicalTargetDir =
        Path.GetDirectoryName(cscbWebPath) + @"\ BuildCSCB";
```

接下来使用 PrecompilationFlags 枚举设置用于编译的标志。PrecompilationFlags 如表 9-2 所列。

表9-2：PrecompilationFlags枚举值

标志值	目　　的
AllowPartiallyTrustedCallers	向构建的程序集添加 APTC 特性
Clean	移除所有现存的编译过的镜像
CodeAnalysis	构建为代码分析
Default	使用默认编译选项
DelaySign	延迟签署程序集
FixedNames	生成的程序集带有用于页的固定名称。不执行批处理编译，只单个进行编译
ForceDebug	确保为调试进行程序集编译
OverwriteTarget	如目标程序集存在，则覆盖原程序集
Updateable	确保程序集是可更新的

要构建一个调试映像并确保在编译没问题时被成功创建，可使用 ForceDebug 和 OverwriteTarget 标志，代码如下所示。

```
PrecompilationFlags flags = PrecompilationFlags.ForceDebug |
                            PrecompilationFlags.OverwriteTarget;
```

然后，将 PrecompilationFlags 存储在 ClientBuildManagerParameter 类的一个新实例中，并使用已经为之设置的参数创建 ClientBuildManager。为完成预构建，要调用 PrecompileApplication 方法。要注意，有一个名为 MyClientBuildManagerCallback 类的实例，它被传递给 PrecompileApplication 方法，代码如下所示。

```
    ClientBuildManagerParameter cbmp = new ClientBuildManagerParameter();
    cbmp.PrecompilationFlags = flags;

    ClientBuildManager cbm =
        new ClientBuildManager(appVirtualDir,
                               appPhysicalSourceDir,
                               appPhysicalTargetDir,
                               cbmp);
    MyClientBuildManagerCallback myCallback = new MyClientBuildManagerCallback();
    cbm.PrecompileApplication(myCallback);
}
```

MyClientBuildManagerCallback 类从 ClientBuildManagerCallback 类派生而来，允许代码在 Web 应用程序编译期间接收通知。ClientBuildManagerCallback 类的方法带有

LinkDemands，要求回调方法也带有它们。编译器错误、解析错误和进度通知都是可用的。在 MyClientBuildManagerCallback 类中，它们皆被实现为写入到调试流和控制台，代码如下所示。

```
public class MyClientBuildManagerCallback : ClientBuildManagerCallback
{
    public MyClientBuildManagerCallback()
        : base()
    {
    }

    [PermissionSet(SecurityAction.Demand, Unrestricted = true)]
    public override void ReportCompilerError(CompilerError error)
    {
        string msg = $"Report Compiler Error: {error.ToString()}";
        Debug.WriteLine(msg);
        Console.WriteLine(msg);
    }

    [PermissionSet(SecurityAction.Demand, Unrestricted = true)]
    public override void ReportParseError(ParserError error)
    {
        string msg = $"Report Parse Error: {error.ToString()}";
        Debug.WriteLine(msg);
        Console.WriteLine(msg);
    }

    [PermissionSet(SecurityAction.Demand, Unrestricted = true)]
    public override void ReportProgress(string message)
    {
        string msg = $"Report Progress: {message}";
        Debug.WriteLine(msg);
        Console.WriteLine(msg);
    }
}
```

来自 CSCB 网站的成功编译输出如下所示。

```
Report Progress: Building directory '/CSCBWeb/Properties'.
Report Progress: Building directory '/CSCBWeb'.
```

9.6.3 讨论

ClientBuildManager 实际上是 BuildManager 类的瘦包装器，BuildManager 类完成编译的大部分工作。ClientBuildManager 能够更直接地确保 Web 应用程序的所有重要部分都得到处理，而 BuildManager 提供了更细粒度的控制。ClientBuildManager 还允许订阅诸如启动、关闭、卸载等应用程序域通知事件，允许在预构建期间应用程序域消失的事件中进行错误处理。

要在 ASP.NET 中不借助 ClientBuildManager 而预构建应用程序，可将一个 HTTP 请求按 http://server/webapp/precompile.xsd 格式发布到网站上。precompile.axd“文档”触发一个将为用户预构建网站的 ASP.NET HttpHandler。这一过程通过 aspnet_compiler.exe 模块处理，

该模块本质上包含了 `ClientBuildManager` 的功能。

9.6.4 参考

MSDN 文档中的“`ClientBuildManager`”“`ClientBuildManagerParameters`”“`BuildManager`”和“ASP.NET Web Site Precompilation”主题。

9.7 为Web应用对数据进行转义和取消转义

9.7.1 问题

你需要将用于 Web 操作的数据从转义格式转换为取消转义格式，或者执行相反的过程以用于正确的传输。这种转义和取消转义应当遵循 RFC 2396-Uniform Resource Identifiers (URI): Generic Syntax 中描述的格式。

9.7.2 解决方案

使用 `Uri` 类上用于转义和取消转义数据与 `Uri` 的静态方法。

要转义数据，可使用静态 `Uri.EscapeDataString` 方法，如下所示。

```
string data = "<H1>My html</H1>";
Console.WriteLine($"Original Data: {data}");
Console.WriteLine();

string escapedData = Uri.EscapeDataString(data);
Console.WriteLine($"Escaped Data: {escapedData}");
Console.WriteLine();

// 上述代码的输出为
// Original Data: <H1>My html</H1>
//
// Escaped Data: %3CH1%3EMy%20html%3C%2FH1%3E
```

要取消转义数据，可使用静态 `Uri.UnescapeDataString` 方法，代码如下所示。

```
string unescapedData = Uri.UnescapeDataString(escapedData);
Console.WriteLine($"Unescaped Data: {unescapedData}");
Console.WriteLine();

// 上述代码的输出为
//
// Unescaped Data: <H1>My html</H1>
```

要转义一个 `Uri`，可使用静态 `Uri.EscapeUriString` 方法，代码如下所示。

```
string uriString = "http://user:password@localhost:8080/www.abc.com/" +
    "home page.htm?item=1233;html=<h1>Heading</h1>#stuff";
Console.WriteLine($"Original Uri string: {uriString}");
Console.WriteLine();
```

```
string escapedUriString = Uri.EscapeUriString(uriString);
Console.WriteLine($"Escaped Uri string: {escapedUriString}");
Console.WriteLine();

// 上述代码的输出为
//
// Original Uri string: http://user:password@localhost:8080/www.abc.com/home
// page.htm?item=1233;html=<h1>Heading</h1>#stuff
//
// Escaped Uri string: http://user:password@localhost:8080/www.abc.com/home
// %20page.htm?item=1233;html=%3Ch1%3EHeading%3C/h1%3E#stuff
```

如果你想知道转义一个 Uri 为何拥有其自己的方法（EscapeUriString），可以在其上使用 Uri.EscapeDataString 和 Uri.UnescapeDataString，看一下被转义的 Uri 是什么样，代码如下所示。

```
// 为什么不直接使用EscapeDataString来转义一个Uri？ 它不够挑剔……
string escapedUriData = Uri.EscapeDataString(uriString);
Console.WriteLine($"Escaped Uri data: {escapedUriData}");
Console.WriteLine();

Console.WriteLine(Uri.UnescapeDataString(escapedUriString));

// 上述代码的输出为
//
// Escaped Uri data: http%3A%2F%2Fuser%3Apassword%40localhost%3A8080%2Fwww.abc.
// com%2Fhome%20page.htm%3Fitem%3D1233%3Bhtml%3D%3Ch1%3EHeading%3C%2Fh1%3E%23
// stuff

// http://user:password@localhost:8080/www.abc.com/home page.htm?item=1233;html
// =<h1>Heading</h1>#stuff
```

我们注意到，:、/、:、@ 和 ? 字符在不应当转义的时候被转义，这就是为什么针对 Uri 要使用 EscapeUriString 方法的原因。

9.7.3 讨论

EscapeUriString 假设在要被转义的字符串中不存在转义序列。转义遵循 RFC 2396 中所制定的约定，将所有保留字符和值大于 128 的字符转换为其十六进制格式。

RFC 2396 的 2.2 节中声明的保留字符包括以下这些。

```
;|/| ? |:| @ | & | = | + | $ | ,
```

在创建一个 System.Uri 对象时，EscapeUriString 方法对于确保 Uri 被正确转义是非常有用的。

9.7.4 参考

MSDN 文档中的“EscapeUriString 方法”“EscapeUriData 方法”和“UnescapeDataString 方法”主题。

9.8　检查Web服务器的自定义错误页

9.8.1　问题

你有一个应用程序，它需要知道给定的 IIS 服务器上设置了哪些针对各种 HTTP 错误返回码的自定义错误页。

9.8.2　解决方案

使用 `System.DirectoryServices.DirectoryEntry` 类与 Internet Information Server（IIS）元数据库交互，以找出建立了哪些自定义错误页。元数据库为 Web 服务器保存配置信息。通过针对 `DirectoryEntry` 的构造函数指定“IIS”模式，`DirectoryEntry` 使用 Active Directory IIS 服务提供程序与元数据库进行通信，代码如下所示。

```
// 这个元数据库中的一个区分大小写的条目
// 你也许以为它拼错了,但是你错了……
const string WebServerSchema = "IIsWebServer";

// 设置为与本地IIS服务器通信
string server = "localhost";

// 使用假的用户名和密码创建一个IIS服务器目录条目
// 如果你以一个常规用户运行,则需要提供凭据
using (DirectoryEntry w3svc =
    new DirectoryEntry($"IIS://{server}/w3svc",
            "Domain/UserCode", "Password"))
{
```

一旦连接建立，那么 Web 服务器模式项被指定于显示 IIS 设置被保存的位置（`IIsWebServer`）。`DirectoryEntry` 有一个属性，允许对其子项（`Children`）进行访问，并且针对每个项的 `SchemaClassName` 都会被检查，以确定它是否属于 Web 服务器设置部分。一旦找到 Web 服务器设置，那么 Web 根节点即可被定位，`HttpErrors` 属性可经由此处获取。`HttpErrors` 是一个以逗号分隔的字符串，表示 HTTP 错误码、HTTP 子错误码、消息类型以及指向当错误发生时返回请求端的 HTTP 文件的路径。为完成这些，只需编写一个 LINQ 查询获得所有的 `HttpErrors`，如例 9-2 所示。只要 `HttpErrors` 被获取，就可以使用 `Split` 方法将它分为一个字符串数组，允许代码访问每一个值并将值写出。执行这些操作的代码如例 9-2 所示。

例 9-2：查找自定义错误页

```
// 使用常规查询表达式以
// 选择机器上所有站点的http错误
var httpErrors = from site in w3svc?.Children.OfType<DirectoryEntry>()
                 where site.SchemaClassName == WebServerSchema
                 from siteDir in site.Children.OfType<DirectoryEntry>()
                 where siteDir.Name == "ROOT"
                 from httpError in siteDir.Properties["HttpErrors"].OfType<string>()
                 select httpError;
```

```
// 使用急切求值将结果转换为数组
// 以便于在每个迭代中不需要重新查询
// 我们将会错过执行中元数据的更新
// 但这相对于重新查询的成本只是很小的代价
// 这将强制查询立即进行一次求值
string[] errors = httpErrors.ToArray();
foreach (var httpError in errors)
{
    //400,*,FILE,C:\WINDOWS\help\iisHelp\common\400.htm
    string[] errorParts = httpError.ToString().Split(',');
    Console.WriteLine("Error Mapping Entry:");
    Console.WriteLine($"\tHTTP error code: {errorParts[0]}");
    Console.WriteLine($"\tHTTP sub-error code: {errorParts[1]}");
    Console.WriteLine($"\tMessage Type: {errorParts[2]}");
    Console.WriteLine($"\tPath to error HTML file: {errorParts[3]}");
}
```

无需使用 LINQ 去查询元数据库当然也可以完成这一工作，如例 9-3 所示。

例 9-3：不使用 LINQ 查找自定义错误页

```
foreach (DirectoryEntry site in w3svc?.Children)
{
    if (site != null)
    {
        using (site)
        {
            // 检查机器上的所有Web服务器
            if (site.SchemaClassName == WebServerSchema)
            {
                // 获得此服务器的元数据库条目
                string metabaseDir = $"/w3svc/{site.Name}/ROOT";

                if (site.Children != null)
                {
                    // 查找每一服务器的ROOT目录
                    foreach (DirectoryEntry root in site.Children)
                    {
                        using (root)
                        {
                            // 我们是否找到了此站点的根目录
                            if (root?.Name.Equals("ROOT",
                                    StringComparison.OrdinalIgnoreCase) ?? false)
                            {
                                // 获得HttpErrors
                                if (root?.Properties.Contains("HttpErrors") == true)
                                {
                                    // 输出
                                    PropertyValueCollection httpErrors =
                                            root?.Properties["HttpErrors"];
                                    for (int i = 0; i < httpErrors?.Count; i++)
                                    {
                                        //400,*,FILE,
                                        //C:\WINDOWS\help\iisHelp\common\400.htm
                                        string[] errorParts =
                                            httpErrors?[i].ToString().Split(',');
```

```
                                Console.WriteLine("Error Mapping Entry:");
                                Console.WriteLine($"\tHTTP error code:" +
                                    $"{errorParts[0]}");
                                Console.WriteLine($"\tHTTP sub-error code:" +
                                    $"{errorParts[1]}");
                                Console.WriteLine($"\tMessage Type: " +
                                    $"{errorParts[2]}");
                                Console.WriteLine(
                                    $"\tPath to error HTML file:
                                    {errorParts[3]}");
                            }
                        }
                    }
                }
            }
        }
    }
}
}
}
```

至此，应用程序可以缓存这些映射自己错误结果的设置，或者可以动态地修改错误页，以提供自定义的内容。此处的结构是，用于 Web 服务器的设置信息对于所有应用程序是轻松可用的，只需要一小点代码。

9.8.3 讨论

`System.DirectoryServices.DirectoryEntry` 通常用于 Active Directory 编程，但它也能够使用任何可用于 Active Directory 的提供程序。这一方法允许代码检查 IIS 元数据库，不管是较老的 IIS 5.x 元数据库还是 Windows Server 自带的较新 IIS 元数据库。

在例 9-2 中，LINQ 被用于查询元数据库，一些有趣的事情发生了。查询遍历了元数据层次以获取 `HttpErrors`，但应当注意到，`DirectoryEntry.Children` 属性是一个 `DirectoryEntries` 集合类。`DirectoryEntries` 确实支持 `IEnumerable`，但它不支持 `IEnumerable<T>`，而后者是 LINQ 工作所需的。`OfType<DirectoryEntry>` 扩展方法从 `DirectoryEntries` 支持的 `IEnumerable` 接口中返回强类型的 `IEnumerable<DirectoryEntry>`。这样做可以找出网站和根目录，然后可以使用 `OfType<string>` 获得一个带有 `HttpErrors` 的字符串的枚举列表。

```
var httpErrors = from site in w3svc?.Children.OfType<DirectoryEntry>()
                 where site.SchemaClassName == WebServerSchema
                 from siteDir in site.Children.OfType<DirectoryEntry>()
                 where siteDir.Name == "ROOT"
                 from httpError in
                     siteDir.Properties["HttpErrors"].OfType<string>()
                 select httpError;
```

我们使用了常规的查询表达式语法编写查询，但我们也可以使用被称为显式点标记（explicit dot notation）的语法来建立查询，如下所示。

```
var httpErrors = w3svc?.Children.OfType<DirectoryEntry>()
                    .Where(site => site.SchemaClassName == WebServerSchema)
                    .SelectMany(siteDir =>
                        siteDir.Children.OfType<DirectoryEntry>())
                    .Where(siteDir => siteDir.Name == "ROOT")
                    .SelectMany<DirectoryEntry, string>(siteDir =>
                    siteDir.Properties["HttpErrors"].OfType<string>());
```

显式点标记语法只是直接从集合类型或已扩展的接口上调用扩展方法，LINQ 正是建立在这些扩展方法之上。这些扩展方法在 System.Core 程序集中的 System.Linq 命名空间下的静态 Enumerable 类上定义，是查询表达式语法构建的基础。查询表达式语法告诉 C# 编译器使用这些扩展方法来执行所请求的查询。

通过使用多个 from 语句，可以将 SelectMany 的使用隐含在通常的查询语法中。SelectMany 允许查询将结果折叠到一个单独的集合中，从而得到 IEnumerable<string> 作为 httpErrors 结果。如果使用 Select，那么结果可能会是 IEnumerable<IEnumerable<string>>；它是一个字符串集合的集合，而不是一个连续的集合。

要构建第一处的查询，更容易的方法是从单独的较小查询入手，然后组合它们。当使用显式点标记语法时，使用下列子查询可以容易地重新进行组合。

```
// 使用显式点标记语法拆散查询,首先获得站点,
// 然后获得http错误属性值

var sites = w3svc?.Children.OfType<DirectoryEntry>()
            .Where(child => child.SchemaClassName == WebServerSchema)
            .SelectMany(child => child.Children.OfType<DirectoryEntry>());

var httpErrors = sites
                 .Where(site => site.Name == "ROOT")
                 .SelectMany<DirectoryEntry,string>(site =>
                     site.Properties["HttpErrors"].OfType<string>());

// 使用显式点标记语法组合查询
var combinedHttpErrors = w3svc?.Children.OfType<DirectoryEntry>()
                         .Where(site => site.SchemaClassName == WebServerSchema)
                         .SelectMany(siteDir =>
                              siteDir.Children.OfType<DirectoryEntry>())
                         .Where(siteDir => siteDir.Name == "ROOT")
                         .SelectMany<DirectoryEntry, string>(siteDir =>
                            siteDir.Properties["HttpErrors"].OfType<string>());
```

9.8.4 参考

MSDN 文档中的“SelectMany<TSource, TResult> 方法”“OfType<TResult> 方法”“HttpErrors [IIS]”“IIS Metabase Properties”和“DirectoryEntry 类”主题。

9.9 编写一个TCP服务器

9.9.1 问题

你需要创建一个服务器，以安全或非安全的方式在一个端口上侦听来自 TCP 客户端的进入请求。然后在服务器端处理这些客户端请求，并将任何响应发送回客户端。范例 9.10（即 9.10 节）展示了如何编写一个与该服务器交互的 TCP 客户端。

9.9.2 解决方案

使用这里创建的 `MyTcpServer` 类，在基于 TCP 的端点侦听到达给定端口的请求，代码如下所示。

```
class MyTcpServer
{
    #region Private Members
    private TcpListener _listener;
    private IPAddress _address;
    private int _port;
    private bool _listening;
    private string _sslServerName;
    private object _syncRoot = new object();
    #endregion

    #region CTORs

    public MyTcpServer(IPAddress address, int port, string sslServerName = null)
    {
        _port = port;
        _address = address;
        _sslServerName = sslServerName;
    }
    #endregion // CTORs
```

`MyTcpServer` 类有以下四个属性。

- `Address`，一个 `IPAddress`
- `Port`，一个 `int`
- `Listening`，一个 `bool`
- `SSLServerName`，一个 `string`

这些属性返回服务器正在侦听的当前地址和端口，侦听状态，以及 `MyTcpServer` 侦听的 SSL（secure sockets layer）服务器名称。

```
#region Properties
public IPAddress Address { get; }

public int Port { get; }

public bool Listening { get; private set; }
```

```
    public string SSLServerName  { get; }
    #endregion
```

ListenAsync 方法告诉 MyTcpServer 类开始侦听指定的地址和端口组合。生成并启动一个 TcpListener，然后运行一个 Task 调用它的 AcceptTcpClientAsync 方法以等待将要到达的某个客户端请求。一旦客户端连接上，运行 ProcessClientAsync 方法以服务客户端交互。

服务完客户端后侦听器关闭。

```
    #region Public Methods
    public async Task ListenAsync(CancellationToken cancellationToken =
 default(CancellationToken))
    {
        cancellationToken.ThrowIfCancellationRequested();
        try
        {
            lock (_syncRoot)
            {
                _listener = new TcpListener(Address, Port);

                // 启动服务器
                _listener.Start();

                // 设置侦听标志
                Listening = true;
            }

            // 进入侦听循环
            do
            {
                Console.Write("Looking for someone to talk to... ");
                // 等待连接
                try
                {
                    cancellationToken.ThrowIfCancellationRequested();
                    await Task.Run(async () =>
                    {
                        TcpClient newClient =
                            await _listener.AcceptTcpClientAsync();
                        Console.WriteLine("Connected to new client");
                        await ProcessClientAsync(newClient, cancellationToken);
                    },cancellationToken);
                }
                catch (OperationCanceledException)
                {
                    // 用户取消
                    Listening = false;
                }
            }
            while (Listening);
        }
        catch (SocketException se)
        {
```

```
            Console.WriteLine($"SocketException: {se}");
        }
        finally
        {
            // 关闭它
            StopListening();
        }
    }
```

调用 StopListening 方法以停止 MyTCPServer 侦听请求，代码如下所示。

```
    public void StopListening()
    {
        if (Listening)
        {
            lock (_syncRoot)
            {
                // 设置侦听标志
                Listening = false;
                try
                {
                    // 如果在侦听,则关闭它
                    if (_listener.Server.IsBound)
                        _listener.Stop();
                }
                catch (ObjectDisposedException)
                {
                    // 如果我们尝试在AcceptTcpClientAsync中
                    // 等待一个连接时(因为它是阻塞操作)停止侦听,
                    // 它将引发一个ObjectDisposedException异常,
                    // 因为在此处我们知道我们正在关闭
                    // 只需提示我们取消了侦听
                    Console.WriteLine("Cancelled the listener");
                }
            }
        }
    }
    #endregion
```

例 9-4 所示的 ProcessClientAsync 方法执行以服务于一个连接的客户端。它确定是否已设置 SSL 连接的服务器名称；如果已设置，使用 TcpClient.GetStream 创建一个 SslStream，并使用配置的服务器名称获取服务器证书。然后使用 AuthenticateAsServer 方法进行身份验证。如果不使用 SSL，ProcessClientAsync 会使用 TcpClient.GetStream 方法从客户端获得 NetworkStream，然后读取整个请求。发送回一个响应之后，该方法关闭客户端连接。

例 9-4：ProcessClientAsyn 方法

```
    #region Private Methods
    private async Task ProcessClientAsync(TcpClient client,
          CancellationToken cancellationToken = default(CancellationToken))
    {
        cancellationToken.ThrowIfCancellationRequested();
        try
        {
            // 用于读取数据的缓冲区
```

```
byte[] bytes = new byte[1024];
StringBuilder clientData = new StringBuilder();

Stream stream = null;
if (!string.IsNullOrWhiteSpace(SSLServerName))
{
    Console.WriteLine($"Talking to client over SSL using {SSLServerName}");
    SslStream sslStream = new SslStream(client.GetStream());
    sslStream.AuthenticateAsServer(GetServerCert(SSLServerName), false,
         SslProtocols.Default, true);
    stream = sslStream;
}
else
{
    Console.WriteLine("Talking to client over regular HTTP");
    stream = client.GetStream();
}
// 获得流以与客户端通信
using (stream)
{
    // 将初始读取超时设置为1分钟以允许连接
    stream.ReadTimeout = 60000;
    // 循环以获取客户端发送的所有数据
    int bytesRead = 0;
    do
    {
        // 这看起来像是一个bug,但它显然不是……
        // 当我们使用Read方法时,第一次工作正常,
        // 然后第二次读取没有数据时,
        // 因NetworkStream上设置的0.5秒超时而引发IOException
        // 如果我们使用ReadAsync,第二次读取没有数据时将永远挂起
        // 这是因为使用Async时,Socket类上的超时将被忽略
        try
        {
            // 此处使用Read而不是ReadAsync,因为如果你调用ReadAsync,
            // 将不会如你所预期的那样发生超时(参见上面的注解)
            bytesRead = stream.Read(bytes, 0, bytes.Length);
            if (bytesRead > 0)
            {
                // 将数据字节转换为ASCII字符串并附加到clientData
                clientData.Append(
                    Encoding.ASCII.GetString(bytes, 0, bytesRead));
                // 既然数据已经到来,将读取超时缩减为1/2秒
                stream.ReadTimeout = 500;
            }
        }
        catch (IOException ioe)
        {
            // 读取超时,所有数据已被获取
            Trace.WriteLine($"Read timed out: {ioe}");
            bytesRead = 0;
        }
    }
    while (bytesRead > 0);
```

```
                Console.WriteLine($"Client says: {clientData}");

                // 感谢他们的输入
                bytes = Encoding.ASCII.GetBytes("Thanks call again!");

                // 发送回响应
                await stream.WriteAsync(bytes, 0, bytes.Length, cancellationToken);
            }
        }
        finally
        {
            // 停止与客户端的交互
            client?.Close();
        }
    }
```

最后，将 MyTCPServer 设置为使用 SSL 时，GetServerCert 方法会获取 X509Certificate。该方法要求本地计算机上个人证书存储中的证书是可访问的。如果它是一个自签名证书，则该证书需要在受信任的根证书存储区中都可用。

```
        private static X509Certificate GetServerCert(string subjectName)
        {
            using (X509Store store =
               new X509Store(StoreName.My, StoreLocation.LocalMachine))
            {
                store.Open(OpenFlags.ReadOnly);
                X509CertificateCollection certificate =
                    store.Certificates.Find(X509FindType.FindBySubjectName,
                       subjectName, true);

                if (certificate.Count > 0)
                    return (certificate[0]);
                else
                    return (null);
            }
        }
    }
```

下面是一个简单服务器的示例，该服务器在按下 Escape 键之前一直侦听客户端。

```
    class Program
    {
        private static MyTcpServer _server;
        private static CancellationTokenSource _cts;

        static void Main()
        {
            _cts = new CancellationTokenSource();
            try
            {
                // 我们并不等待这一调用,因为我们想要继续执行
                // 以使得控制台UI可以处理按键
                RunServer(_cts.Token);
            }
            catch(Exception ex)
```

```
            {
                Console.WriteLine(ex.ToString());
            }
            string msg = "Press Esc to stop the server...";
            Console.WriteLine(msg);
            ConsoleKeyInfo cki;
            while (true)
            {
                cki = Console.ReadKey();
                if (cki.Key == ConsoleKey.Escape)
                {
                    _cts.Cancel();
                    _server.StopListening();
                    break; // 允许退出
                }
            }
            Console.WriteLine("");
            Console.WriteLine("All done listening");
        }

        private static async Task RunServer(CancellationToken cancellationToken)
        {
            try
            {
                await Task.Run(async() =>
                {
                    cancellationToken.ThrowIfCancellationRequested();
                    _server = new MyTcpServer(IPAddress.Loopback, 55555);
                    await _server.ListenAsync(cancellationToken);
                }, cancellationToken);
            }
            catch (OperationCanceledException)
            {
                Console.WriteLine("Cancelled.");
            }
        }
    }
```

当与范例 9.10（即 9.10 节）中的 `MyTcpClient` 类对话时，服务器的输出如下所示。

```
Press Esc to stop the server...
Looking for someone to talk to... Connected to new client
Client says: Just wanted to say hi
Looking for someone to talk to... Connected to new client
Client says: Just wanted to say hi again
Looking for someone to talk to... Connected to new client
Client says: Are you ignoring me?
Looking for someone to talk to... Connected to new client
Client says: I'll not be ignored! (round 0)
Looking for someone to talk to... Connected to new client
Client says: I'll not be ignored! (round 1)
Looking for someone to talk to... Connected to new client
Client says: I'll not be ignored! (round 2)
Looking for someone to talk to... Connected to new client
Client says: I'll not be ignored! (round 3)
```

```
Looking for someone to talk to... Connected to new client
Client says: I'll not be ignored! (round 4)
Looking for someone to talk to... Connected to new client
Client says: I'll not be ignored! (round 5)
Looking for someone to talk to... Connected to new client
Client says: I'll not be ignored! (round 6)
Looking for someone to talk to... Connected to new client
Client says: I'll not be ignored! (round 7)
Looking for someone to talk to... Connected to new client
Client says: I'll not be ignored! (round 8)
Looking for someone to talk to... Connected to new client
Client says: I'll not be ignored! (round 9)
Looking for someone to talk to... Connected to new client
Client says: I'll not be ignored! (round 10)
    [more output follows...]
```

9.9.3 讨论

传输控制协议（TCP）是目前互联网上大多数通信所使用的协议。TCP 负责从一端到另一端正确地传递数据包。它使用互联网协议（IP）执行传递。IP 处理节点间的包获取，TCP 检测包何时不正确、丢失或无序发送，并且安排重发丢失或者损坏的包。MyTCPServer 类是一种基本的服务器机制，用来处理 TCP 上客户端的请求。

MyTcpServer 获得传递到构造函数的 IP 地址和端口，并且在该 IPAddress 和端口上创建一个 TcpListener。一旦创建完毕，就调用 TcpListener.Start 方法启动服务器。AcceptTcpClientAsync 方法被调用以侦听来自基于 TCP 的客户端请求并等待来自客户端的连接。一旦客户端连接上，将执行 ProcessClientAsyn 方法。在这个方法中，服务器读取来自客户端的请求数据并返回一个简要回执。服务器端通过调用 TcpClient.Close 与客户端断开连接。当调用 StopListening 方法时，服务器停止。StopListening 通过调用 TcpListener.Stop 使服务器离线。

要支持安全的请求，你可以在 MyTCPServer 构造函数中设置 SSLServerName，用于标识要使用的身份验证的证书。

运行服务器的程序然后在类构造函数中提供此名称，如下所示。

```
_server = new MyTcpServer(IPAddress.Loopback, 55555, "CSharpCookBook.net");
```

在 ListenAsync 方法中，我们使用了 lock 语句，代码如下所示。

```
public async Task ListenAsync(CancellationToken cancellationToken =
    default(CancellationToken))
{
    cancellationToken.ThrowIfCancellationRequested();
    try
    {
        lock (_syncRoot)
        {
            _listener = new TcpListener(Address, Port);

            // 启动服务器
```

```
        _listener.Start();

        // 设置侦听标志
        Listening = true;
    }
```

MSDN 中将 lock 定义为如下：lock 关键字将语句块标记为临界区，方法是获取给定对象的互斥锁，执行语句，然后释放该锁。虽然这是事实，你可以更简单地认为："没有其他线程将运行 lock 语句放在方括号内的代码部分，直到第一个线程完成。"那些喜欢挑战极限的人可能会想："嘿，我可以在 lock 语句内使用 async 和 await，然后它将让行给下一个线程，对吗？"是的，从技术上讲可以，但是你不应该这样做，因为它几乎肯定会在你的应用程序中导致死锁。你 await 的代码可能执行了 lock 而导致死锁。lock 内部的代码也可能恢复到另一个线程（因为当你 await，它通常不会恢复到同一线程上），所以你会从不同的线程中解锁，而不是建立锁的那个线程。这是一个"非常坏的事情"，所以请不要这样做。

9.9.4 参考

MSDN 文档中的"IPAddress 类""TcpListener 类""SslStream 类""lock 语句"和"TcpClient 类"主题。

9.10 编写一个TCP客户端

9.10.1 问题

你希望以安全或非安全的方式与一个基于 TCP 的服务器交互。

9.10.2 解决方案

使用例 9-5 所示的 MyTcpClient 类，传入要对话的服务器地址、端口和 SSL 服务器名（如果已进行身份验证），使用 System.Net.TcpClient 类与一个基于 TCP 的服务器连接并对话。该示例将与范例 9.9（即 9.9 节）中的服务器对话。

例 9-5：MyTcpClient 类

```
class MyTcpClient : IDisposable
{
    private TcpClient _client;
    private IPEndPoint _endPoint;
    private bool _disposed;

    #region Properties
    public IPAddress Address { get; }

    public int Port { get; }
```

```
public string SSLServerName { get; }

#endregion

public MyTcpClient(IPAddress address, int port, string sslServerName = null)
{
    Address = address;
    Port = port;
    _endPoint = new IPEndPoint(Address, Port);
    SSLServerName = sslServerName;
}

public async Task ConnectToServerAsync(string msg)
{
    try
    {
        _client = new TcpClient();
        await _client.ConnectAsync(_endPoint.Address,_endPoint.Port);

        Stream stream = null;
        if (!string.IsNullOrWhiteSpace(SSLServerName))
        {
            SslStream sslStream =
                new SslStream(_client.GetStream(), false,
                    new RemoteCertificateValidationCallback(
                        CertificateValidationCallback));
            sslStream.AuthenticateAsClient(SSLServerName);
            DisplaySSLInformation(SSLServerName, sslStream, true);
            stream = sslStream;
        }
        else
        {
            stream = _client.GetStream();
        }
        using (stream)
        {
            // 获得要发送消息的字节数据
            byte[] bytes = Encoding.ASCII.GetBytes(msg);

            // 发送消息
            Console.WriteLine($"Sending message to server: {msg}");
            await stream?.WriteAsync(bytes, 0, bytes.Length);

            // 获得响应
            // 用于保存响应字节数据的缓冲区
            bytes = new byte[1024];

            // 显示响应
            int bytesRead = await stream?.ReadAsync(bytes, 0, bytes.Length);
            string serverResponse =
                Encoding.ASCII.GetString(bytes, 0, bytesRead);
            Console.WriteLine($"Server said: {serverResponse}");
        }
    }
```

```
    catch (SocketException se)
    {
        Console.WriteLine($"There was an error talking to the server: {se}");
    }
    finally
    {
        Dispose();
    }
}

#region IDisposable Members

public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}

private void Dispose(bool disposing)
{
    if (!_disposed)
    {
        if (disposing)
        {
            _client?.Close();
        }
        _disposed = true;
    }
}

#endregion

private bool CertificateValidationCallback(object sender,
        X509Certificate certificate,
        X509Chain chain,
        SslPolicyErrors sslPolicyErrors)
{
    if (sslPolicyErrors == SslPolicyErrors.None)
    {
        return true;
    }
    else
    {
        if (sslPolicyErrors == SslPolicyErrors.RemoteCertificateChainErrors)
        {
            Console.WriteLine("The X509Chain.ChainStatus returned an array of " +
                "X509ChainStatus objects containing error information.");
        }
        else if (sslPolicyErrors ==
            SslPolicyErrors.RemoteCertificateNameMismatch)
        {
            Console.WriteLine(
                "There was a mismatch of the name on a certificate.");
        }
        else if (sslPolicyErrors ==
```

```
            SslPolicyErrors.RemoteCertificateNotAvailable)
        {
            Console.WriteLine("No certificate was available.");
        }
        else
        {
            Console.WriteLine("SSL Certificate Validation Error!");
        }

        Console.WriteLine("");
        Console.WriteLine("SSL Certificate Validation Error!");
        Console.WriteLine(sslPolicyErrors.ToString());

        return false;
    }
}

private static void DisplaySSLInformation(string serverName,
    SslStream sslStream, bool verbose)
{
    DisplayCertInformation(sslStream.RemoteCertificate, verbose);

    Console.WriteLine("");
    Console.WriteLine($"SSL Connect Report for : {serverName}");
    Console.WriteLine("");
    Console.WriteLine(
        $"Is Authenticated:            {sslStream.IsAuthenticated}");
    Console.WriteLine($"Is Encrypted:                {sslStream.IsEncrypted}");
    Console.WriteLine($"Is Signed:                   {sslStream.IsSigned}");
    Console.WriteLine($"Is Mutually Authenticated:   " +
        $"{sslStream.IsMutuallyAuthenticated}");
    Console.WriteLine("");
    Console.WriteLine($"Hash Algorithm:              {sslStream.HashAlgorithm}");
    Console.WriteLine($"Hash Strength:               {sslStream.HashLength}");
    Console.WriteLine(
        $"Cipher Algorithm:            {sslStream.CipherAlgorithm}");
    Console.WriteLine(
        $"Cipher Strength:             {sslStream.CipherStrength}");
    Console.WriteLine("");
    Console.WriteLine($"Key Exchange Algorithm:      " +
        $"{sslStream.KeyExchangeAlgorithm}");
    Console.WriteLine($"Key Exchange Strength:       " +
        $"{sslStream.KeyExchangeStrength}");
    Console.WriteLine("");
    Console.WriteLine($"SSL Protocol:                {sslStream.SslProtocol}");
}

private static void DisplayCertInformation(X509Certificate remoteCertificate,
    bool verbose)
{
    Console.WriteLine("");
    Console.WriteLine("Certificate Information for:");
    Console.WriteLine($"{remoteCertificate.Subject}");
    Console.WriteLine("");
    Console.WriteLine("Valid From:");
```

```
            Console.WriteLine($"{remoteCertificate.GetEffectiveDateString()}");
            Console.WriteLine("Valid To:");
            Console.WriteLine($"{remoteCertificate.GetExpirationDateString()}");
            Console.WriteLine("Certificate Format:");
            Console.WriteLine($"{remoteCertificate.GetFormat()}");
            Console.WriteLine("");
            Console.WriteLine("Issuer Name:");
            Console.WriteLine($"{remoteCertificate.Issuer}");

            if (verbose)
            {
                Console.WriteLine("Serial Number:");
                Console.WriteLine($"{remoteCertificate.GetSerialNumberString()}");
                Console.WriteLine("Hash:");
                Console.WriteLine($"{remoteCertificate.GetCertHashString()}");
                Console.WriteLine("Key Algorithm:");
                Console.WriteLine($"{remoteCertificate.GetKeyAlgorithm()}");
                Console.WriteLine("Key Algorithm Parameters:");
                Console.WriteLine(
                    $"{remoteCertificate.GetKeyAlgorithmParametersString()}");
                Console.WriteLine("Public Key:");
                Console.WriteLine($"{remoteCertificate.GetPublicKeyString()}");
            }
        }
    }
```

要在程序中使用 MyTcpClient，你可以简单地生成它的一个实例并调用 ConnectToServerAsync 发送一个请求。在 TalkToServerAsync 方法中，首先对服务器端执行三个调用，以测试基本机制，并等待 MakeClientCallToServer 方法的结果。接下来，进入一个循环在其上真正运行并创建多个 Task 请求，各自等待 MakeClientCallToServerAsync 方法。这验证了服务器端处理多个请求的机制是合理的。

```
    static void Main()
    {
        Task serverChat = TalkToServerAsync();
        serverChat.Wait();
        Console.WriteLine(@"Press the ENTER key to continue...");
        Console.Read();
    }

    private static async Task MakeClientCallToServerAsync(string msg)
    {
        MyTcpClient client = new MyTcpClient(IPAddress.Loopback, 55555);
        // 取消注释以使用SSL与服务器对话
        //MyTcpClient client = new MyTcpClient(IPAddress.Loopback, 55555,
        //    "CSharpCookBook.net");
        await client.ConnectToServerAsync(msg);
    }

    private static async Task TalkToServerAsync()
    {
        await MakeClientCallToServerAsync("Just wanted to say hi");
        await MakeClientCallToServerAsync("Just wanted to say hi again");
        await MakeClientCallToServerAsync("Are you ignoring me?");
```

```
        // 现在发送一批消息
        string msg;
        for (int i = 0; i < 100; i++)
        {
            msg = $"I'll not be ignored! (round {i})";
            RunClientCallAsTask(msg);
        }
    }

    private static void RunClientCallAsTask(string msg)
    {
        Task work = Task.Run(async () =>
        {
            await MakeClientCallToServerAsync(msg);
        });
    }
```

用于这些消息交换的客户端输出如下所示。

```
Sending message to server: Just wanted to say hi
Server said: Thanks call again!
Sending message to server: Just wanted to say hi again
Server said: Thanks call again!
Sending message to server: Are you ignoring me?
Server said: Thanks call again!
Press the ENTER key to continue...
Sending message to server: I'll not be ignored! (round 1)
Sending message to server: I'll not be ignored! (round 0)
Sending message to server: I'll not be ignored! (round 2)
Sending message to server: I'll not be ignored! (round 3)
Sending message to server: I'll not be ignored! (round 4)
Sending message to server: I'll not be ignored! (round 6)
Sending message to server: I'll not be ignored! (round 5)
Sending message to server: I'll not be ignored! (round 7)
Sending message to server: I'll not be ignored! (round 9)
Sending message to server: I'll not be ignored! (round 10)

[once all requests are set up as tasks you see the responses...]

Server said: Thanks call again!
Server said: Thanks call again!
Server said: Thanks call again!
Server said: Thanks call again!
Server said: Thanks call again!
Server said: Thanks call again!
Server said: Thanks call again!
Server said: Thanks call again!
Server said: Thanks call again!
Server said: Thanks call again!
Server said: Thanks call again!
```

9.10.3 讨论

`MyTcpClient.ConnectToServerAsync` 设计用于发送一条消息，获得响应，显示为一个字符

串，然后关闭连接。为了完成这项工作，它生成一个 System.Net.TcpClient 并通过调用 TcpClient.ConnectAsync 方法与服务器连接。ConnectAsync 通过使用传入 MyTCpClient 构造函数的地址和端口构建一个 IPEndPoint 来指定服务器。

MyTCpClient.ConnectToServerAsync 然后使用 Encoding.ASCII.GetBytes 方法获得对应字符串的字节数据。一旦发送了字节，它便通过调用其 GetStream 方法从底层的 System.Net.TcpClient 中获得 NetworkStream 或者 SslStream，然后使用 TcpClient.WriteAsync 方法发送消息。

为了接收来自服务器端的响应，MyTcpClient.ConnectToServerAsync 调用了阻塞的 TcpClient.ReadAsync 方法。一旦 ReadAsync 返回，字节便被解码以获得包含来自服务器端响应的字符串。然后关闭连接，客户端结束。

要支持安全的请求，可以在 MyTcpClient 构造函数中设置 SSLServerName，用于标识要用于身份验证的证书。

运行客户端的程序然后将此名称提供给类构造函数，如下所示。

```
MyTcpClient client =
    new MyTcpClient(IPAddress.Loopback, 55555, "CSharpCookBook.net");
```

当使用一个安全的连接时，我们使用 MyTcpClient 的 DisplaySSLInformation 和 DisplayCertInformation 方法显示所有连接的细节，因为它们涉及证书和安全状态。

```
Certficate Information for:
CN=CSharpCookBook.net

Valid From:
12/27/2014 7:29:31 PM
Valid To:
12/31/2039 6:59:59 PM
Certificate Format:
X509

Issuer Name:
CN=CSharpCookBook.net
Serial Number:
0F0E1C4148C6A09C42EDEDAFCD2E83E2
Hash:
664E30B62C4FB9DBEE0C29F27A15E5EDE2C46187
Key Algorithm:
1.2.840.113549.1.1.1
Key Algorithm Parameters:
0500
Public Key:
3082020A0282020100EAB6004CD3F2F5214773E8FE4FA40FE610F1C27E888276E81EBBB86020B904
3B136CF02197C928ED0BCA8339A31334059C2744A8BB617849BBC98C8B242FC360C88BF62E2C491B
1A6F951DDB65E0036D8839AC6695B26CD3E50DD749A5610C8564CF99EE79FED272D04A3100B51A4A
4BAE076BB8129E39B382ED1FDB8382A2D3C057D7F46072DDDE0654083E1F2CB4E25685B5EE4B4F25
F3D2561B61869D9C39B9FB389E6A06D9DEFA6693D94C6A1F2CA34462B3D9C68CF91A179B0957050E
A9A30D508C067C216CAD59CA9E846B0EBA02472333BBF2462415B13567EBF6930FC1000EECC3EA70
9867B8BD6869BF828B8EBA5BA2E4A7660B46B798A8BB8D046FFE1C767F5A77AF1CD6E83F9E013AB1
748264F89617D9C106813F554B8AF4184AC58B55A1A58ABAA2F171CDBFF6923C27FE801FEE5D3664
```

```
87F54FAD184B0FCBB874532EC8E6B3BAA322F05DB6AD99E5982B98AD43C0E9BB2356270DB07BA5E5
AAE2F0B66E630A6A0435FDFC61DB46B0FF348AF5D2285C74A35E8AAFC86F45C0E674C2D9FE98B6C1
17208668CF4B03DD77948AE45AE84D33178C3042B1155E58D3B49492697D5CA4CF4AB24549E4A240
CCEB6CF61CEF6F33F412A91BC32803136A6481B6B246FEA5A3943EEB7FDA5E54CC561DE737BBB380
BC2B467F1A5B8CA1BDFC66B6B4E60DCCC7C3912449D0BF8B9878D22C04A36A09898D2AAED0CE32DB
770203010001

SSL Connect Report for : CSharpCookBook.net

Is Authenticated:            True
Is Encrypted:                True
Is Signed:                   True
Is Mutually Authenticated:   False

Hash Algorithm:              Sha1
Hash Strength:               160
Cipher Algorithm:            Aes256
Cipher Strength:             256

Key Exchange Algorithm:      44550
Key Exchange Strength:       256

SSL Protocol:                Tls
Sending message to server: I'll not be ignored! (round 95)
Server said: Thanks call again!
```

在 9.10.2 节中，我们在 MyTCpClient 中添加的 IDisposable 接口的实现如下所示。

```
#region IDisposable Members

public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}

private void Dispose(bool disposing)
{
    if (!_disposed)
    {
        if (disposing)
        {
            _client?.Close();
        }
        _disposed = true;
    }
}

#endregion
```

我们这样做是为了正确地处理私有的 TcpClient 实例变量 _client 的关闭，因为它提供了自己的 Close 方法，以便执行一些日志记录并清理其资源。在 Dispose 方法中调用了 SuppressFinalize 以通知垃圾回收器该对象已被完全清理。

9.10.4　参考

MSDN 文档中的“TcpClient 类”“SslStream 类”“NetworkStream 类”“IDisposable 接口”和“Encoding.ASCII 属性”主题。

9.11　模拟表单执行

9.11.1　问题

你需要发送一个名称 / 值对集合到一个 URL 所指定的位置，以模拟一个表单在浏览器中的执行。

9.11.2　解决方案

使用 System.Net.WebClient 类的 UploadValues 方法向 Web 服务器发送一组名称 / 值对。该类通过使用输入数据设置名称 / 值对，使得你假装成执行一个表单的浏览器。输入域的 ID 是名称，而域中使用的值是值。

```
// 为了使用此代码,首先需要运行CSCBWeb项目
Uri uri = new Uri("http://localhost:4088/WebForm1.aspx");
WebClient client = new WebClient();

// 创建一系列名称/值对以发送
// 将必要的参数/值对添加到名称/值容器中
NameValueCollection collection = new NameValueCollection()
    { {"Item", "WebParts"},
        {"Identity", "foo@bar.com"},
        {"Quantity", "5"} };

Console.WriteLine(
    $"Uploading name/value pairs to URI {uri.AbsoluteUri} ...");

// 上传NameValueCollection
byte[] responseArray =
    await client.UploadValuesTaskAsync(uri, "POST", collection);

// 解码并显示响应
Console.WriteLine(
    $"\nResponse received was {Encoding.UTF8.GetString(responseArray)}");
```

接收和处理该数据的 WebForm1.aspx 页面如下所示。

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="WebForm1.aspx.cs"
    Inherits="WebForm1" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
```

```
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>


        <asp:Table ID="Table1" runat="server" Height="139px" Width="361px">
            <asp:TableRow runat="server">
                <asp:TableCell runat="server"><asp:Label ID="Label1"
                runat="server"
Text="Identity"></asp:Label></asp:TableCell>
                <asp:TableCell runat="server"><asp:TextBox ID="Identity"
runat="server"/></asp:TableCell>
            </asp:TableRow>
            <asp:TableRow runat="server">
                <asp:TableCell runat="server"><asp:Label ID="Label2"
                runat="server"
Text="Item"></asp:Label></asp:TableCell>
                <asp:TableCell runat="server"><asp:TextBox ID="Item"
runat="server"/></asp:TableCell>
            </asp:TableRow>
            <asp:TableRow runat="server">
                <asp:TableCell runat="server"><asp:Label ID="Label3"
                runat="server"
Text="Quantity"></asp:Label></asp:TableCell>
                <asp:TableCell runat="server"><asp:TextBox ID="Quantity"
runat="server"/></asp:TableCell>
            </asp:TableRow>
            <asp:TableRow runat="server">
                <asp:TableCell runat="server"></asp:TableCell>
                <asp:TableCell runat="server"><asp:Button ID="Button1"
                runat="server"
onclick="Button1_Click" Text="Submit" /></asp:TableCell>
            </asp:TableRow>
        </asp:Table>

    </div>
    </form>
</body>
</html>
```

背后的 WebForm1.aspx.cs 代码如下所示。

```
using System;
using System.Web;

public partial class WebForm1 : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if(HttpContext.Current.Request.HttpMethod.ToUpper() == "POST")
            WriteOrderResponse();
    }
    protected void Button1_Click(object sender, EventArgs e)
```

```
    {
        WriteOrderResponse();
    }

    private void WriteOrderResponse()
    {
        string response = "Thanks for the order!<br/>";
        response += "Identity: " + Request.Form["Identity"] + "<br/>";
        response += "Item: " + Request.Form["Item"] + "<br/>";
        response += "Quantity: " + Request.Form["Quantity"] + "<br/>";
        Response.Write(response);
    }
}
```

表单执行的输出如下所示。

```
Uploading name/value pairs to URI http://localhost:4088/WebForm1.aspx ...

Response received was ?Thanks for the order!<br/>Identity: foo@bar.com<br/>Item:
 WebParts<br/>Quantity: 5<br/>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.or
g/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head><title>
        Untitled Page
</title></head>
<body>
    <form name="form1" method="post" action="WebForm1.aspx" id="form1">
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value="/wEPDwULLTE3NDA4
NzI1OTJkZHS2esbeFu36oKf1n3XvCfLBFbminq7tuASWazSmVzNV" />

    <div>


                <table id="Table1" border="0" height="139" width="361">
        <tr>
                <td><span id="Label1">Identity</span></td><td><input name="Ident
ity" type="text" id="Identity" /></td>
        </tr><tr>
                <td><span id="Label2">Item</span></td><td><input name="Item" typ
e="text" id="Item" /></td>
        </tr><tr>
                <td><span id="Label3">Quantity</span></td><td><input name="Quant
ity" type="text" id="Quantity" /></td>
        </tr><tr>
                <td></td><td><input type="submit" name="Button1" value="Submit"
id="Button1" /></td>
        </tr>
</table>

    </div>

<input type="hidden" name="__VIEWSTATEGENERATOR" id="__VIEWSTATEGENERATOR" value
```

```
="B6E7D48B" />
<input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION" value="/wEd
AAWO/dj0xplxW6YoKRXH5OHbmz/pl7ppA227nN6820C6Sskwyhj63BXMkV5ahbRAQpWWUallXbdbKxLN
IxdB86x+zfg78Z8BXhXifTCAVkevd657ebmKYjtae5uEq9PVWd0RhH/uhX8f6dI/Hiyy1p14" /></fo
rm>

<!-- Visual Studio Browser Link -->
<script type="application/json" id="__browserLink_initializationData">
    {"appName":"Unknown","requestId":"c7ee16b51c9b4bccae0c3c79a9fba779"}
</script>
<script type="text/javascript" src="http://localhost:2976/eef9532a4f984be0b28884
3bb4cee559/browserLink" async="async"></script>
<!-- End Browser Link -->

</body>
</html>
```

9.11.3 讨论

WebClient 类使得以名称 / 值对的常用格式将表单数据上传到 Web 服务器变得很简单。你可以通过使用一个 URI（http://localhost:4088/WebForm1.aspx），选用的 HTTP 方法（POST）和你创建的 NameValueCollection（collection）调用 UploadValuesTaskAsync 来了解该技术。

> UploadValues* 方法的异步版本被调用，并且所用的方法（UploadValuesTaskAsync）是用于 async 和 await 的特定方法。

通过调用其 Add 方法，传递输入域的 id 作为名称以及放入域中的值作为值，NameValueCollection 被填入表单中每个域中的数据。在本示例中，你在 Identity 域中填入 foo@bar.com，在 Item 域中填入 Book，在 Quantity 域中填入 5。然后在控制台窗口中输出来自 POST 的响应结果。

9.11.4 参考

MSDN 文档中的“WebClient 类”主题。

9.12 通过HTTP传输数据

9.12.1 问题

你需要从一个 URL 指定的位置上下载或上传数据，该数据可能是一个字节数组或者一个文件。

9.12.2 解决方案

使用 WebClient.UploadDataTaskAsync 或 WebClient.DownloadDataTaskAsync 方法使用一个 URL 传输数据。

要从一个网页下载数据，请执行以下操作。

```
Uri uri = new Uri("http://localhost:4088/DownloadData.aspx");

// 创建一个客户端
using (WebClient client = new WebClient())
{
    // 获得文件内容
    Console.WriteLine($"Downloading {uri.AbsoluteUri}");
    // 下载页面并保存字节数据
    byte[] bytes;
    try
    {
        // 注意,还有一个DownloadDataAsync方法,用于旧的EAP模式,
        // 此处我们不会用到
        bytes = await client.DownloadDataTaskAsync(uri);
    }
    catch (WebException we)
    {
        Console.WriteLine(we.ToString());
        return;
    }
    // 输出HTML
    string page = Encoding.UTF8.GetString(bytes);
    Console.WriteLine(page);

}
```

这将产生下列输出。

```
Downloading http://localhost:4088/DownloadData.aspx
?

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.or
g/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head><title>
        Download Data
</title></head>
<body>
    <form name="Form1" method="post" action="DownloadData.aspx" id="Form2">
        <input type="hidden" name="__VIEWSTATE"
value="dDwyMDQwNjUzNDY2Ozs+kS9hguYm9369sybDqmIow0AvxBg=" />
    <span id="Label1" style="Z-INDEX: 101; LEFT: 142px; POSITION: absolute;
TOP: 164px">This is downloaded html!</span>
    </form>

<!-- Visual Studio Browser Link -->
<script type="application/json" id="__browserLink_initializationData">
```

```
    {"appName":"Unknown","requestId":"b43b962ff6264058b5dbf17aed23a082"}
</script>
<script type="text/javascript" src="http://localhost:3587/db7b63d3424649c7a10386
29bc71b103/browserLink" async="async"></script>
<!-- End Browser Link -->

</body>
</html>
```

你还可以使用 DownloadFileTaskAsync 将数据下载到一个文件中。

```
Uri uri = new Uri("http://localhost:4088/DownloadData.aspx");

// 创建一个客户端
using (WebClient client = new WebClient())
{
    // 去获得文件
    Console.WriteLine($"Retrieving file from {uri}...{Environment.NewLine}");
    // 获得文件并放入一个临时文件中
    string tempFile = Path.GetTempFileName();
    try
    {
        // 注意,还有一个DownloadFileAsync方法,用于旧的EAP模式,
        // 此处我们不会用到
        await client.DownloadFileTaskAsync(uri, tempFile);
    }
    catch (WebException we)
    {
        Console.WriteLine(we.ToString());
        return;
    }
    Console.WriteLine($"Downloaded {uri} to {tempFile}");
}
```

这将产生下列输出（临时文件路径和名称将会有所不同）。

```
Retrieving file from http://localhost:4088/DownloadData.aspx...

Downloaded http://localhost:4088/DownloadData.aspx to C:\Users\jhilyard\AppData\
Local\Temp\tmpA5D7.tmp
```

要将一个文件上传到一个 URL 中，可使用 UploadFileTaskAsync，代码如下所示。

```
Uri uri = new Uri("http://localhost:4088/UploadData.aspx");
// 创建一个客户端
using (WebClient client = new WebClient())
{
    Console.WriteLine($"Uploading to {uri.AbsoluteUri}");
    try
    {
        // 注意,还有一个UploadFileAsync方法,用于旧的EAP模式,
        // 此处我们不会用到
        await client.UploadFileTaskAsync(uri, "SampleClassLibrary.dll");
        Console.WriteLine($"Uploaded successfully to {uri.AbsoluteUri}");
    }
    catch (WebException we)
```

```
    {
        Console.WriteLine(we.ToString());
    }
}
```

可以接收上传文件的 ASPX 页面代码如下所示。

```
using System;
using System.Web;

public partial class UploadData : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        foreach (string f in Request.Files.AllKeys)
        {
            HttpPostedFile file = Request.Files[f];
            // 需要有写入目录的写权限
            try
            {
                string path = Server.MapPath(".") + @"\" + file.FileName;
                file.SaveAs(path);
                Response.Write("Saved " + path);
            }
            catch (HttpException hex)
            {
                // 返回保存文件的特定错误信息
                Response.Write("Failed to save file with error: " +
                    hex.Message);
            }
        }
    }
}
```

虽然上述 ASPX 页面将接收和存储文件，这只是为了说明使用 WebClient 上传的基本示例。当构建能够接收文件的页面时，请确保处理了文件上传安全方面的问题，如 OWASP（开放式 Web 应用程序安全项目）在其网站（https://www.owasp.org/index.php/Unrestricted_File_Upload）上描述的未限制的文件上传漏洞。

这将产生如下输出。

```
Uploading to http://localhost:4088/UploadData.aspx
Uploaded successfully to http://localhost:4088/UploadData.aspx
```

9.12.3 讨论

WebClient 简化了文件和文件中字节的下载，因为在处理 Web 时它们都是常见的任务。更传统的用于下载的基于流的方法也可以通过 WebClient 上的 OpenReadTaskAsync 方法访问。

9.12.4 参考

MSDN 文档中的“WebClient 类”主题和 OWASP 网站（https://www.owasp.org/）。

9.13 使用命名管道进行通信

9.13.1 问题

你需要一种使用命名管道通过网络与另一应用程序进行通信的方法。

9.13.2 解决方案

使用 System.IO.Pipes 命名空间中的 NamedPipeClientStream 和 NamedPipeServerStream。然后可以创建一个客户端和服务器端来使用命名管道。

为了使用 NamedPipeClientStream 类，你需要类似于例 9-6 中所示的某些代码。

例 9-6：使用 NamedPipeClientStream 类

```
using System;
using System.Text;
using System.IO.Pipes;
using System.Threading.Tasks;

namespace NamedPipes
{
    class NamedPipeClientConsole
    {
        static void Main()
        {
            Task client = RunClient();
            client.Wait();

            Console.WriteLine("Press Enter to exit...");
            Console.ReadLine();
        }

        private static async Task RunClient()
        {
            Console.WriteLine("Initiating client, looking for server...");
            // 设置一个要发送的消息
            string messageText = "Sample text message!";
            int bytesRead;

            // 设置一个命名管道客户端,并在完成之后关闭它
            using (NamedPipeClientStream clientPipe =
                    new NamedPipeClientStream(".", "mypipe", PipeDirection.InOut,
                    PipeOptions.None))
            {
                // 连接至服务器的流
                await clientPipe.ConnectAsync();
                // 将读取模式设置为message
```

```
                clientPipe.ReadMode = PipeTransmissionMode.Message;

                // 写入十次消息
                for (int i = 0; i < 10; i++)
                {
                    Console.WriteLine($"Sending message: {messageText}");
                    byte[] messageBytes = Encoding.Unicode.GetBytes(messageText);
                    // 检查并写入消息
                    if (clientPipe.CanWrite)
                    {
                        await clientPipe.WriteAsync(
                            messageBytes, 0, messageBytes.Length);
                        await clientPipe.FlushAsync();
                        // 等待直到被读取
                        clientPipe.WaitForPipeDrain();
                    }

                    // 设置一个用于消息字节的缓冲区
                    messageBytes = new byte[256];
                    do
                    {
                        // 将消息收集到字符串生成器中
                        StringBuilder message = new StringBuilder();

                        // 读取所有数据,直到获得
                        // 完整的消息响应
                        do
                        {
                            // 从管道中读取
                            bytesRead =
                                await clientPipe.ReadAsync(
                                    messageBytes, 0, messageBytes.Length);
                            // 如果获得了数据,将其添加到消息中
                            if (bytesRead > 0)
                            {
                                message.Append(
                                    Encoding.Unicode.GetString(messageBytes, 0,
                                        bytesRead));
                                Array.Clear(messageBytes, 0, messageBytes.Length);
                            }
                        }
                        while (!clientPipe.IsMessageComplete);

                        // 既然已读取了整条消息,将其设置为0
                        bytesRead = 0;
                        Console.WriteLine($"    Received message: " +
                            $"{message.ToString()}");
                    }
                    while (bytesRead != 0);
                }
            }

        }
    }
}
```

然后，为了建立一个与客户端通话的服务器，可使用 `NamedPipeServerStream` 类，如例 9-7 所示。

例 9-7：为客户端建立一个服务器

```
using System;
using System.Text;
using System.IO.Pipes;
using System.Threading.Tasks;

namespace NamedPipes
{
    class NamedPipeServerConsole
    {
        static void Main()
        {
            Task server = RunServer();
            server.Wait();

            // 使服务器挂起,以便看到发送的消息
            Console.WriteLine("Press Enter to exit...");
            Console.ReadLine();
        }

        private static async Task RunServer()
        {
            Console.WriteLine("Initiating server, waiting for client...");
            // 以message模式启动命名管道,并在完成之后关闭管道
            using (NamedPipeServerStream serverPipe = new
                    NamedPipeServerStream("mypipe", PipeDirection.InOut, 1,
                    PipeTransmissionMode.Message, PipeOptions.None))
            {
                // 等待一个客户端……
                await serverPipe.WaitForConnectionAsync();

                // 处理消息,直到客户端断开连接
                while (serverPipe.IsConnected)
                {
                    int bytesRead = 0;
                    byte[] messageBytes = new byte[256];
                    // 读取,直到获得了消息,然后进行响应
                    do
                    {
                        // 构建客户端消息
                        StringBuilder message = new StringBuilder();

                        // 检查是否可读取管道
                        if (serverPipe.CanRead)
                        {
                            // 循环直到整条消息被读取
                            do
                            {
                                bytesRead =
                                    await serverPipe.ReadAsync(messageBytes, 0,
                                                  messageBytes.Length);
```

```
                        // 从流中获得了字节,所以添加到消息中
                        if (bytesRead > 0)
                        {
                            message.Append(
                                Encoding.Unicode.GetString(messageBytes, 0,
                                    bytesRead));
                            Array.Clear(messageBytes, 0,
                                messageBytes.Length);
                        }
                    }
                    while (!serverPipe.IsMessageComplete);
                }

                // 如果我们收到了一条消息,输出它然后回应
                if (message.Length > 0)
                {
                    // 既然已读到整条消息,将其设置为0
                    bytesRead = 0;
                    Console.WriteLine($"Received message: " +
                        $"{message.ToString()}");

                    // 翻转从客户端收到的消息文本并返回
                    char[] messageChars =
                        message.ToString().Trim().ToCharArray();
                    Array.Reverse(messageChars);
                    string reversedMessageText = new string(messageChars);

                    // 显示返回的消息
                    Console.WriteLine($"    Returning Message: " +
                        $"{{reversedMessageText}");

                    // 输出回应
                    messageBytes = Encoding.Unicode.GetBytes(messageChars);
                    if (serverPipe.CanWrite)
                    {
                        // 输出消息
                        await serverPipe.WriteAsync(messageBytes, 0,
                            messageBytes.Length);
                        // 刷新缓冲区
                        await serverPipe.FlushAsync();
                        // 等待客户端读取
                        serverPipe.WaitForPipeDrain();
                    }
                }
            }
            while (bytesRead != 0);
        }
    }

  }
 }
}
```

9.13.3 讨论

命名管道是 Windows 上一种允许进程间或机器间进行通信的机制。.NET Framework 中提供了对命名管道的托管访问，这使得在托管应用程序中使用命名管道变得更加简单。在许多情况下，你可以使用 Windows Communication Foundation（WCF）建立服务器和客户端代码，WCF 甚至提供了一个命名管道绑定以完成这一工作。这取决于用户应用程序需求以及你希望在应用程序栈的哪个级别上工作。如果你有一个现存的建立了命名管道的应用程序，那么当你能够直接连接时为什么要使用 WCF 呢？使用命名管道类似于使用套接字并保持代码贴近于管道。这样做的好处在于需处理的代码层次较少，缺点在于在消息处理方面必须做更多的工作。

在解决方案中，我们创建了某些使用 `NamedPipeClientStream` 和 `NamedPipeServerStream` 的代码。它们之间的交互如下所示。

(1) 服务器进程启动，它创建一个 `NamedPipeClientStream`，然后调用 `WaitForConnectionAsync` 等待某一客户端来连接：

```
// 以message模式启动命名管道,并在完成之后关闭管道
using (NamedPipeServerStream serverPipe = new
           NamedPipeServerStream("mypipe", PipeDirection.InOut, 1,
               PipeTransmissionMode.Message, PipeOptions.None))
{
     // 等待一个客户端……
    await serverPipe.WaitForConnectionAsync();
```

(2) 客户端进程被创建，它创建一个 `NamedPipeClientStream`，调用 `ConnectAsync` 并与服务器进程连接：

```
// 设置一个命名管道客户端,并在完成之后关闭它
using (NamedPipeClientStream clientPipe =
     new NamedPipeClientStream(".","mypipe",
           PipeDirection.InOut,PipeOptions.None))
{
    // 连接至服务器的流
    await clientPipe.ConnectAsync();
```

(3) 服务器端进程看到来自客户端的连接，然后在一个查找来自客户端消息的循环中调用 `IsConnected`，直到连接断开：

```
// 处理消息,直到客户端断开连接
while (serverPipe.IsConnected)
{
      // 此处略掉更多的处理代码……
}
```

(4) 客户端进程然后使用 `WriteAsync`、`FlushAsync` 和 `WaitForPipeDrain` 向服务器端进程写入若干消息：

```
string messageText = "Sample text message!";

// 写入十次消息
for (int i = 0; i < 10; i++)
```

```
{
    Console.WriteLine($"Sending message: {messageText}");
    byte[] messageBytes = Encoding.Unicode.GetBytes(messageText);
    // 检查并写入消息
    if (clientPipe.CanWrite)
    {
        await clientPipe.WriteAsync(
            messageBytes, 0, messageBytes.Length);
        await clientPipe.FlushAsync();
        // 等待直到被读取
        clientPipe.WaitForPipeDrain();
    }
    // 响应处理……
}
```

(5) 当客户端进程接收到来自服务器的响应时，它读取消息字节直至全部完成。如果消息发送完成，那么 NamedPipeClientStream 跳出 using 语句的作用域并关闭（因此关闭客户端的连接），然后等待用户按下回车键以便退出：

```
// 设置一个用于消息字节的缓冲区
messageBytes = new byte[256];
do
{
    // 将消息收集到字符串生成器中
    StringBuilder message = new StringBuilder();

    // 读取所有数据,直到获得完整的消息响应
    do
    {
        // 从管道中读取
        bytesRead =
            await clientPipe.ReadAsync(
                messageBytes, 0, messageBytes.Length);
        // 如果获得了数据,将其添加到消息中
        if (bytesRead > 0)
        {
            message.Append(
                    Encoding.Unicode.GetString(messageBytes, 0,
                        bytesRead));
            Array.Clear(messageBytes, 0, messageBytes.Length);
        }
    }
    while (!clientPipe.IsMessageComplete);

    // 既然已读取了整条消息,将其设置为0
    bytesRead = 0;
    Console.WriteLine($"    Received message: {message.ToString()}");
}
while (bytesRead != 0);
```

(6) 服务器进程通过 white 循环中失败的 IsConnected 调用注意到客户端已经关闭了管道连接。NamedPipeServerStream 跳出 using 语句的作用域，并关闭管道。

客户端输出如下所示。

```
Initiating client, looking for server...
Sending message: Sample text message!
    Received message: !egassem txet elpmaS
Sending message: Sample text message!
    Received message: !egassem txet elpmaS
Sending message: Sample text message!
    Received message: !egassem txet elpmaS
Sending message: Sample text message!
    Received message: !egassem txet elpmaS
Sending message: Sample text message!
    Received message: !egassem txet elpmaS
Sending message: Sample text message!
    Received message: !egassem txet elpmaS
Sending message: Sample text message!
    Received message: !egassem txet elpmaS
Sending message: Sample text message!
    Received message: !egassem txet elpmaS
Sending message: Sample text message!
    Received message: !egassem txet elpmaS
Sending message: Sample text message!
    Received message: !egassem txet elpmaS
Press Enter to exit...
```

服务器输出如下列示。

```
Initiating server, waiting for client...
Received message: Sample text message!
    Returning Message: !egassem txet elpmaS
Received message: Sample text message!
    Returning Message: !egassem txet elpmaS
Received message: Sample text message!
    Returning Message: !egassem txet elpmaS
Received message: Sample text message!
    Returning Message: !egassem txet elpmaS
Received message: Sample text message!
    Returning Message: !egassem txet elpmaS
Received message: Sample text message!
    Returning Message: !egassem txet elpmaS
Received message: Sample text message!
    Returning Message: !egassem txet elpmaS
Received message: Sample text message!
    Returning Message: !egassem txet elpmaS
Received message: Sample text message!
    Returning Message: !egassem txet elpmaS
Received message: Sample text message!
    Returning Message: !egassem txet elpmaS
Press Enter to exit...
```

`PipeOptions` 枚举控制管道操作如何运行。枚举值如表 9-3 所示。

表9-3：PipeOptions枚举值

成员名	描　述
None	未指定特定选项

（续）

成员名	描　　述
WriteThrough	当向管道写入时，操作不返回控制，直至在服务器端完成写入。没有该标志，写入会被缓冲，并且写入的返回更快速
Asychronous	启用异步管道使用（调用立即返回并在后台进行处理）

9.13.4　参考

MSDN 文档中的“命名管道”“NamedPipeClientStream 类”“NamedPipeServerStream 类”和“System.IO.Pipes 命名空间”主题。

9.14　以编程方式发送ping

9.14.1　问题

你希望检查网络中一台计算机的可用性。

9.14.2　解决方案

使用 System.Net.NetworkInformation.Ping 类确定一台机器是否可用。在 TestPing 方法中，创建 Ping 类的一个实例。使用 Send 方法发送一个 ping 请求。SendPingAsync 方法是异步的，并且在 await 等待后返回一个可检查 ping 结果的 PingReply。也可以在挂接 Ping 类的 PingCompleted 事件之后，异步地使用旧的 SendAsync 方法执行第二个 ping 请求。SendAsync 方法的第二个参数保存了一个用户令牌值，当 ping 结束时返回到 pinger_PingCompleted 事件处理程序。

在 async 和 await 对你可用时应当使用 SendPingAsync，但是如果你在旧的框架中进行此操作，那么 SendAsync 是你唯一的异步选项。返回的令牌可用于识别初始代码和结束代码之间的请求。

```
public static async Task TestPing()
{
    System.Net.NetworkInformation.Ping pinger =
        new System.Net.NetworkInformation.Ping();
    PingReply reply = await pinger.SendPingAsync("www.oreilly.com");
    DisplayPingReplyInfo(reply);

    pinger.PingCompleted += pinger_PingCompleted;
    pinger.SendAsync("www.oreilly.com", "oreilly ping");
}
```

DisplayPingReplyInfo 方法展示了你希望从 ping 中获得的更常用的数据片段，诸如 RoundtripTime 和回复的 Status。这些可通过 PingReply 上的那些属性访问。

```
private static void DisplayPingReplyInfo(PingReply reply)
{
```

```
        Console.WriteLine("Results from pinging " + reply.Address);
        Console.WriteLine(
            $"\tFragmentation allowed?: {!reply.Options.DontFragment}");
        Console.WriteLine($"\tTime to live: {reply.Options.Ttl}");
        Console.WriteLine($"\tRoundtrip took: {reply.RoundtripTime}");
        Console.WriteLine($"\tStatus: {reply.Status.ToString()}");
    }
```

用于 `PingCompleted` 的事件处理程序是 `pinger_PingCompleted` 方法。该事件处理程序遵循发送方对象和事件参数的一般 `EventHandler` 约定。该事件的参数类型是 `PingCompletedEventArgs`。`PingReply` 可通过事件参数的 `Reply` 属性访问。如果 ping 被取消或者引发一个异常，那么该信息可通过 `Cancelled` 和 `Error` 属性访问。`PingCompletedEventArgs` 类上的 `UserState` 属性持有 `SendAsync` 中提供的用户令牌值。

```
    private static void pinger_PingCompleted(object sender, PingCompletedEventArgs e)
    {
        PingReply reply = e.Reply;
        DisplayPingReplyInfo(reply);

        if (e.Cancelled)
            Console.WriteLine($"Ping for {e.UserState.ToString()} was cancelled");
        else
            Console.WriteLine(
                $"Exception thrown during ping: {e.Error?.ToString()}");
    }
```

`DisplayPingReplyInfo` 的输出如下所示。

```
    Results from pinging 23.3.106.121
            Fragmentation allowed?: True
            Time to live: 60
            Roundtrip took: 13
            Status: Success
```

9.14.3　讨论

`Ping` 使用一个 RFC 792 定义的互联网控制消息协议（ICMP）中的 echo 请求消息。如果 ping 请求未能成功到达一台计算机，这并不能意味着该计算机是不可访问的。除了机器离线之外，还有许多因素会阻止一个 ping 成功到达。网络拓扑、防火墙、包过滤器以及代理服务器都可能中断 ping 请求的正常流。Windows 防火墙默认禁止 ICMP 通信，因此如果你发送 ping 到一台机器时遇到困难，请检查该机器上的防火墙设置。

9.14.4　参考

MSDN 文档中的“`Ping` 类”“`PingReply` 类”和“`PingCompleted` 事件”主题。

9.15 使用SMTP服务发送SMTP邮件

9.15.1 问题

你希望能够从自己的程序通过 SMTP 发送电子邮件，但不想学习 SMTP 协议并手动编写一个类去实现它。

9.15.2 解决方案

使用 `System.Net.Mail` 命名空间，其中包含了处理构建一个基于 SMTP 的电子邮件消息的较难部分的类。`System.Net.Mail.MailMessage` 类封装了对一个基于 SMTP 的消息的构建，而 `System.Net.Mail.SmtpClient` 类提供了向 SMTP 服务器发送消息的发送机制。`SmtpClient` 依赖在某处建立一个消息中继的 SMTP 服务器。可以通过创建 `System.Net.Mail.Attachment` 的实例并提供指向文件的路径以及媒体类型来添加附件，代码如下所示。

```
// 发送一个包含附件的消息
string from = "authors@oreilly.com";
string to = "authors@oreilly.com";
MailMessage attachmentMessage = new MailMessage(from, to);
attachmentMessage.Subject = "Hi there!";
attachmentMessage.Body = "Check out this cool code!";
// 许多系统过滤通过中继的HTML邮件
attachmentMessage.IsBodyHtml = false;
// 设置附件
string pathToCode = @"..\..\09_NetworkingAndWeb.cs";
Attachment attachment =
    new Attachment(pathToCode,
        MediaTypeNames.Application.Octet);
attachmentMessage.Attachments.Add(attachment);

// 或者仅发送文本
MailMessage textMessage = new MailMessage("authors@oreilly.com",
                    "authors@oreilly.com",
                    "Me again",
                    "You need therapy, talking to yourself is one thing but
writing code to send email is a whole other thing...");
```

要发送一个不带附件的简单电子邮件，可以仅用收件地址、发件地址、主题和正文信息调用 `System.Net.Mail.MailMessage` 构造函数。`MailMessage` 构造函数的这一版本简单地填充这些项目，然后可以将其传递给 `SmtpClient.Send` 进行发送，代码如下所示。

```
// 如果本地存在一个SMTP服务,你就能够从本地SMTP服务发出去
// 本地的SMTP服务需要设置为中继到一个真正的
// 邮件服务器,如同你在IIS6中进行的设置
//SmtpClient client = new SmtpClient("localhost");

// 既然我们生活在一个更加有安全意识的时代,我们可以提供正确的参数
// 以连接到SMTP服务器,包括主机名、
// 端口、已启用的SSL,以及你的凭据
// 注意,如果你没有替换当前值,就会得到一个
```

```
// 如下所示的异常：
// System.Net.Mail.SmtpException: The SMTP host was not found. --->
// System.Net.WebException: The remote name could not be resolved:
// 'YOURSMTPSERVERHERE'
using (SmtpClient client = new SmtpClient("YOURSMTPSERVERHERE", 999))
{
    client.EnableSsl = true;
    client.Credentials = new NetworkCredential("YOURSMTPUSERNAME",
    // "YOURSMTPPASSWORD");
    await client.SendMailAsync(attachmentMessage);
}
```

9.15.3 讨论

如 RFC 821 中所定义，SMTP 代表简单邮件传输协议。为了使用 `System.Net.Mail.SmtpClient` 类来利用 .NET Framework 对 SMTP 的支持，必须指定一个进行消息中继的 SMTP 服务器。旧版本的 Windows（Windows 8/Windows Server 2012 之前版本）中，操作系统带有一个可作为 IIS 的一部分安装的 SMTP 服务器。在 9.15.2 节中，`SmtpClient` 通过为要连接的服务器指定 `"localhost"` 来使用这一功能，代表本地机器是 SMTP 中继服务器。在你的网络环境中建立 SMTP 服务器也许不可能，你可能需要使用 `SmtpClient` 类建立凭据来直接与网络上的 SMTP 服务器连接，如 9.15.2 节所示。

```
using(SmtpClient client = new SmtpClient("YOURSMTPSERVERHERE",999))
{
    client.EnableSsl = true;
    client.Credentials = new NetworkCredential("YOURSMTPUSERNAME",
    // "YOURSMTPPASSWORD");
    await client.SendMailAsync(attachmentMessage);
}
```

9.15.2 节中用到的 `MediaTypeNames` 类确定了附件类型。有效的附件类型如表 9-4 所列。

表9-4：`MediaTypeNames.Attachment`值

名　称	描　述
`Octet`	数据未被指定为任何特定类型
`Pdf`	数据是便携式数据格式（portable data format）
`Rtf`	数据是 RTF 格式（rich text format）
`Soap`	数据是一个 SOAP 文档
`Zip`	数据是被压缩的

9.15.4 参考

MSDN 文档中的“Using SMTP for Outgoing Messages”“`SmtpMail` 类”“`MailMessage` 类”和“`MailAttachment` 类”主题。

9.16 使用套接字扫描机器的端口

9.16.1 问题

你希望确定一台机器上打开的端口，以查看安全风险的位置。

9.16.2 解决方案

为此可使用 `CheapoPortScanner` 类，其代码如例 9-8 所示。`CheapoPortScanner` 使用 `Socket` 类尝试打开一个套接字并连接某一地址上的指定端口。`ScanAsync` 方法支持通过 `IProgress<T>` 报告在 `CheapoPortScanner` 构造函数中指定的端口范围或默认范围（1~65535）内的每个端口的进展。`CheapoPortScanner` 默认将扫描本地机器。

例 9-8：`CheapoPortScanner` 类

```
public class CheapoPortScanner
{
    #region Private consts and members
    private const int PORT_MIN_VALUE = 1;
    private const int PORT_MAX_VALUE = 65535;
    private List<int> _openPorts;
    private List<int> _closedPorts;
    #endregion
```

值得一提的是 `CheapoPortScanner` 上的两个属性。`OpenPorts` 和 `ClosedPorts` 属性返回一个类型为 `int` 的 `ReadOnlyCollection`，分别为包含打开的和关闭的端口列表。其代码如例 9-9 所示。

例 9-9：`OpenPorts` 和 `ClosedPorts` 属性

```
#region Properties
public ReadOnlyCollection<int> OpenPorts =>
    new ReadOnlyCollection<int>(_openPorts);
public ReadOnlyCollection<int> ClosedPorts =>
    new ReadOnlyCollection<int>(_closedPorts);

public int MinPort { get; } = PORT_MIN_VALUE;
public int MaxPort { get; } = PORT_MAX_VALUE;
public string Host { get; } = "127.0.0.1"; // localhost

#endregion // Properties
#region CTORs & Init code
public CheapoPortScanner()
{
    // 端口和本地主机已作为默认设置
    SetupLists();
}

public CheapoPortScanner(string host, int minPort, int maxPort)
{
    if (minPort > maxPort)
        throw new ArgumentException("Min port cannot be greater than max port");
```

```
    if (minPort < PORT_MIN_VALUE || minPort > PORT_MAX_VALUE)
        throw new ArgumentOutOfRangeException(
            $"Min port cannot be less than {PORT_MIN_VALUE} " +
            $"or greater than {PORT_MAX_VALUE}");
    if (maxPort < PORT_MIN_VALUE || maxPort > PORT_MAX_VALUE)
        throw new ArgumentOutOfRangeException(
            $"Max port cannot be less than {PORT_MIN_VALUE} " +
            $"or greater than {PORT_MAX_VALUE}");

    this.Host = host;
    this.MinPort = minPort;
    this.MaxPort = maxPort;
    SetupLists();
}

private void SetupLists()
{
    // 将列表容量设置为范围的一半大小
    // 因为我们并不能知道有多少端口是打开的
    // 所以折中分配足够一半的容量

    // rangeCount 为 max - min + 1
    int rangeCount = (this.MaxPort - this.MinPort) + 1;
    // 如果结果为奇数,增加一个额外的位置
    if (rangeCount % 2 != 0)
        rangeCount += 1;
    // 为范围内的端口保留一半空间
    _openPorts = new List<int>(rangeCount / 2);
    _closedPorts = new List<int>(rangeCount / 2);
}
#endregion // CTORs & Init code

#region Progress Result
public class PortScanResult
{
    public int PortNum { get; set; }

    public bool IsPortOpen { get; set; }
}

#endregion // Progress Result

#region Private Methods
private async Task CheckPortAsync(int port, IProgress<PortScanResult> progress)
{
    if (await IsPortOpenAsync(port))
    {
        // 如果代码执行至此,说明端口是打开的
        _openPorts.Add(port);

        // 通知关注事件的任何人
        progress?.Report(
            new PortScanResult() { PortNum = port, IsPortOpen = true });
    }
    else
```

```
        {
            // 服务器并没有打开该端口
            _closedPorts.Add(port);
            progress?.Report(
                new PortScanResult() { PortNum = port, IsPortOpen = false });
        }
    }

    private async Task<bool> IsPortOpenAsync(int port)
    {
        Socket sock = null;
        try
        {
            // 创建一个基于TCP的套接字
            sock = new Socket(AddressFamily.InterNetwork,
                            SocketType.Stream,
                            ProtocolType.Tcp);
            // 连接
            await Task.Run(() => sock.Connect(this.Host, port));
            return true;
        }
        catch (SocketException se)
        {
            if (se.SocketErrorCode == SocketError.ConnectionRefused)
                return false;
            else
            {
                // 试图访问套接字时发生了一个错误
                Debug.WriteLine(se.ToString());
                Console.WriteLine(se.ToString());
            }
        }
        finally
        {
            if (sock?.Connected ?? false)
                sock?.Disconnect(false);
            sock?.Close();
        }
        return false;
    }
    #endregion
```

CheapoPortScanner 的触发方法是 ScanAsync。ScanAsync 将检查所有处于构造函数中指定范围内的端口。LastPortScanSummary 方法会将最近一次扫描的相关信息转储到控制台输出流，代码如下所示。

```
#region Public Methods
public async Task ScanAsync(IProgress<PortScanResult> progress)
{
    for (int port = this.MinPort; port <= this.MaxPort; port++)
        await CheckPortAsync(port, progress);
}

public void LastPortScanSummary()
```

```
    {
        Console.WriteLine($"Port Scan for host at {this.Host}");
        Console.WriteLine($"\tStarting Port: {this.MinPort}");
        Console.WriteLine($"\tEnding Port: {this.MaxPort}");
        Console.WriteLine($"\tOpen ports: {string.Join(",", _openPorts)}");
        Console.WriteLine($"\tClosed ports: {string.Join(",", _closedPorts)}");
    }

    #endregion // Public Methods
}
```

TestPortScanner 方法通过扫描本地机器上的 75~85 端口来展示了如何使用 CheapoPortScanner。一个 Progress<CheapoPortScanner.PortScanResult> 报告器被创建并使用一个匿名函数订阅其 ProgressChanged 事件以报告扫描的进展。然后，TestPortScan 使用创建的 Progress<T> 来调用 ScanAsync 方法，以在扫描器工作时得到进度报告。最后，调用 LastPortScanSummary 显示扫描的完整结果，其中包含关闭的端口以及打开的端口。

```
public static async Task TestPortScanner()
{
    // 进行特定范围的扫描
    Console.WriteLine("Checking ports 75-85 on localhost...");
    CheapoPortScanner cps =
        new CheapoPortScanner("127.0.0.1", 75, 85);
    var progress = new Progress<CheapoPortScanner.PortScanResult>();
    progress.ProgressChanged += (sender, args) =>
    {
        Console.WriteLine(
            $"Port {args.PortNum} is " +
            $"{args.IsPortOpen ? "open" : "closed"}");
    };
    await cps.ScanAsync(progress);
    cps.LastPortScanSummary();

    // 进行本地机器扫描,包含整个端口范围1~65 535
    //cps = new CheapoPortScanner();
    //await cps.Scan(progress);
    //cps.LastPortScanSummary();
}
```

端口扫描器的输出如下所示。

```
Checking ports 75-85 on localhost...
Port 75 is closed
Port 76 is closed
Port 77 is closed
Port 78 is closed
Port 79 is closed
Port 80 is open
Port 81 is closed
Port 82 is closed
Port 83 is closed
Port 84 is closed
Port 85 is closed
Port Scan for host at 127.0.0.1
```

```
Starting Port: 75
Ending Port: 85
Open ports: 80
Closed ports: 75,76,77,78,79,81,82,83,84,85
```

9.16.3 讨论

一台机器上打开的端口是值得注意的，因为它们表明有程序在侦听那些端口。黑客通过查找“打开的”端口以作为无需授权进入你的系统的方法。CheapoPortScanner 无可否认是一种不完善的用于检查打开端口的机制，但它很好地展示了原理以提供一个好的起点。

> 如果你在一个公司网络中运行它，那么也许会很快被网络管理员询问，因为你很可能引发了一些入侵检测系统中的警报。请审慎使用这些代码。

9.16.4 参考

MSDN 文档中的“Socket 类”和“Sockets”主题。

9.17 使用当前的互联网连接设置

9.17.1 问题

你希望在程序中使用系统当前的互联网连接设置，而无需强制用户手动在应用程序中添加它们。

9.17.2 解决方案

使用例 9-10 中为你提供的 InternetSettingsReader 类读取当前互联网连接设置。InternetSettingsReader 通过 P/Invoke 调用一些 WinINet API 的方法，以获取当前互联网连接信息。

> P/Invoke（平台调用）是 .NET Framework 中为执行本地调用到非托管（不运行在 .NET CLR 中）代码的机制。当你使用 P/Invoke 时，托管和非托管代码之间传递的数据需要跨越边界进行封送。封送是执行不同层间的调用并将参数和返回数据从托管到非托管进行转换，然后再次转换回托管的过程。结构通常用于传输数据集，因为它们是栈上的值类型并且可用作输入 / 输出参数来传输数据；而类是存在于堆上的引用类型，并且通常仅能用于输入参数。

主要工作在于建立 WinINet 使用的结构并正确地封送结构指针以获取值。

例 9-10：InternetSettingsReader 类

```
public class InternetSettingsReader
{
    #region Private Members
    string _proxyAddr;
    int _proxyPort = -1;
    bool _bypassLocal;
    string _autoConfigAddr;
    List<string> _proxyExceptions;
    PerConnFlags _flags;
    #endregion

    #region CTOR
    public InternetSettingsReader()
    {
    }
    #endregion
```

例 9-11 中所示的 InternetSettingsReader 的每个属性都调用了 GetInternetConnectionOption 方法，它返回一个 InternetConnectionOption。InternetConnectionOption 结构持有所有与要返回的值相关的数据，而该值则根据指定属性所要求的类型来获取。

例 9-11：InternetSettingsReader 属性

```
#region Properties
public string ProxyAddress
{
    get
    {
        InternetConnectionOption ico =
            GetInternetConnectionOption(
                PerConnOption.INTERNET_PER_CONN_PROXY_SERVER);
        // 解析地址和端口
        string proxyInfo = Marshal.PtrToStringUni(
                                ico.m_Value.m_StringPtr);
        ParseProxyInfo(proxyInfo);
        return _proxyAddr;
    }
}
public int ProxyPort
{
    get
    {
        InternetConnectionOption ico =
            GetInternetConnectionOption(
                PerConnOption.INTERNET_PER_CONN_PROXY_SERVER);
        // 解析地址和端口
        string proxyInfo = Marshal.PtrToStringUni(
                                ico.m_Value.m_StringPtr);
        ParseProxyInfo(proxyInfo);
        return _proxyPort;
    }
}
public bool BypassLocalAddresses
{
```

```
        get
        {
            InternetConnectionOption ico =
                GetInternetConnectionOption(
                    PerConnOption.INTERNET_PER_CONN_PROXY_BYPASS);
            // bypass在例外列表中以<local>列出
            string exceptions =
                Marshal.PtrToStringUni(ico.m_Value.m_StringPtr);

            if (exceptions.IndexOf("<local>") != -1)
                _bypassLocal = true;
            else
                _bypassLocal = false;
            return _bypassLocal;
        }
    }
    public string AutoConfigurationAddress
    {
        get
        {
            InternetConnectionOption ico =
                GetInternetConnectionOption(
                    PerConnOption.INTERNET_PER_CONN_AUTOCONFIG_URL);
            // 直接获取
            _autoConfigAddr =
                Marshal.PtrToStringUni(ico.m_Value.m_StringPtr);
            if (_autoConfigAddr == null)
                _autoConfigAddr = "";
            return _autoConfigAddr;
        }
    }
    public IList<string> ProxyExceptions
    {
        get
        {
            InternetConnectionOption ico =
                GetInternetConnectionOption(
                    PerConnOption.INTERNET_PER_CONN_PROXY_BYPASS);
            // 例外以分号来分隔
            string exceptions =
                Marshal.PtrToStringUni(ico.m_Value.m_StringPtr);
            if (!string.IsNullOrEmpty(exceptions))
            {
                _proxyExceptions = new List<string>(exceptions.Split(';'));
            }
            return _proxyExceptions;
        }
    }
    public PerConnFlags ConnectionType
    {
        get
        {
            InternetConnectionOption ico =
                GetInternetConnectionOption(
                    PerConnOption.INTERNET_PER_CONN_FLAGS);
```

```
            _flags = (PerConnFlags)ico.m_Value.m_Int;

            return _flags;
        }
    }

    #endregion

    #region Private Methods
    private void ParseProxyInfo(string proxyInfo)
    {
        if (!string.IsNullOrEmpty(proxyInfo))
        {
            string[] parts = proxyInfo.Split(':');
            if (parts.Length == 2)
            {
                _proxyAddr = parts[0];
                try
                {
                    _proxyPort = Convert.ToInt32(parts[1]);
                }
                catch (FormatException)
                {
                    // 没有端口
                    _proxyPort = -1;
                }
            }
            else
            {
                _proxyAddr = parts[0];
                _proxyPort = -1;
            }
        }
    }
```

例 9-12 所示的 GetInternetConnectionOption 方法承担了与 WinINet 通信的大部分工作。首先，创建一个 InternetPerConnOptionList 和一个用于保存返回值的 InternetConnectionOption 结构。然后 InternetConnectionOption 结构被固定，使得垃圾回收器不会在内存中移动结构，并且对 PerConnOption 赋值以确定获取哪个互联网选项。Marshal.SizeOf 用于确定非托管内存中两个托管结构的大小。这些值被用于初始化结构的大小，这允许操作系统确定处理的非托管结构的版本。

InternetPerConnOptionList 被初始化以保存选项值，然后调用 WinInet 函数 InternetQueryOption。InternetConnectionOption 类可使用 Marshal.PtrToStructure 方法填充，它将来自非托管代码的包含 InternetConnectionOption 数据的非托管结构中的值映射到托管对象实例，然后使用该值返回托管版本。

例 9-12：GetInternetConnectionOption 方法

```
private static InternetConnectionOption GetInternetConnectionOption(
    PerConnOption pco)
{
    //分配list和option
```

```
        InternetPerConnOptionList perConnOptList = new InternetPerConnOptionList();
        InternetConnectionOption ico = new InternetConnectionOption();
        //固定option结构
        GCHandle gch = GCHandle.Alloc(ico, GCHandleType.Pinned);
        //用我们想要的数据初始化option
        ico.m_Option = pco;
        //将option list初始化为默认连接或LAN
        int listSize = Marshal.SizeOf(perConnOptList);
        perConnOptList.dwSize = listSize;
        perConnOptList.szConnection = IntPtr.Zero;
        perConnOptList.dwOptionCount = 1;
        perConnOptList.dwOptionError = 0;
        // 确定大小和偏移值
        int icoSize = Marshal.SizeOf(ico);
        // 为option分配足够内存(本地内存,非.NET堆)
        perConnOptList.options =
            Marshal.AllocCoTaskMem(icoSize);

        // 从结构体中创建指针
        IntPtr optionListPtr = perConnOptList.options;
        Marshal.StructureToPtr(ico, optionListPtr, false);

        //创建查询
        if (NativeMethods.InternetQueryOption(
            IntPtr.Zero,
            75, //(int)InternetOption.INTERNET_OPTION_PER_CONNECTION_OPTION,
            ref perConnOptList,
            ref listSize) == true)
        {
            //获取值
            ico =
                (InternetConnectionOption)Marshal.PtrToStructure(
                                        perConnOptList.options,
                                        typeof(InternetConnectionOption));
        }
        // 释放COM内存
        Marshal.FreeCoTaskMem(perConnOptList.options);
        //解除结构的固定
        gch.Free();

        return ico;
    }
    #endregion
}
```

InternetSettingsReader 的使用通过例 9-13 中所示的 GetInternetSettings 方法进行演示。此处获取代理信息并显示到控制台，但该信息可很容易存储在其他程序中，以在连接时用作代理信息。有关为一个 WebRequest 设置代理信息的详细内容，请参考范例 9.3（即 9.3 节）。

例 9-13：使用 InternetSettingsReader

```
    public static void GetInternetSettings()
    {
            Console.WriteLine("");
```

```
            Console.WriteLine("Reading current internet connection settings");
            InternetSettingsReader isr = new InternetSettingsReader();
            Console.WriteLine($"Current Proxy Address: {isr.ProxyAddress}");
            Console.WriteLine($"Current Proxy Port: {isr.ProxyPort}");
            Console.WriteLine($"Current ByPass Local Address setting: " +
                $"{{isr.BypassLocalAddresses}");
            Console.WriteLine("Exception addresses for proxy (bypass):");
            string exceptions;
            if (isr.ProxyExceptions?.Count > 0)
                exceptions = "\t" + (string.Join(",", isr.ProxyExceptions?.ToArray()));
            else
                exceptions = "\tNone";
            Console.WriteLine($"Proxy connection type: {isr.ConnectionType.ToString()}");
            Console.WriteLine("");
    }
```

解决方案的输出如下所示。

```
Reading current internet connection settings
Current Proxy Address: http=127.0.0.1
Current Proxy Port: -1
Current ByPass Local Address setting: False
Exception addresses for proxy (bypass):
        <-loopback>
Proxy connection type: PROXY_TYPE_DIRECT
```

9.17.3 讨论

WinInet Windows Internet（WinInet）API 是用于与 FTP、HTTP 和 Gopher 协议进行交互的非托管 API。该 API 可用于提供托管代码尚未提供的功能，例如 9.17.2 节所示的互联网配置设置。它还用于下载文件、使用 Cookie 以及参与 Gopher 会话。要记住 WinInet 是一个客户端的 API，它不适用于服务器端或服务应用程序，不正确的使用会引发应用程序中的问题。

直接通过 BCL（基础类库）可为 C# 程序员提供大量可用的信息，但有时仍然需要迎难而上地与 Win32 API 对话。即使是在有限特权为规范的情况下，创建一个需要增强访问以进行 P/Invoke 的小程序集也常常是合理的。它可能会被锁定访问，以避免成为对系统的风险。我们在范例 11.6（即 11.6 节）中展示了如何限制一个程序集，你需要使用 `SecurityPermissionFlag.UnmanagedCode` 来断言 `SecurityPermission`。

9.17.4 参考

MSDN 文档中的“InternetQueryOption function [WinInet]”“与非托管代码交互操作”和“Using P/Invoke to Call Unmanaged APIs from Your Managed Classes”主题。

9.18 使用FTP传输文件

9.18.1 问题

你希望以编程方式使用文件传输协议（FTP）下载和上传文件。

9.18.2 解决方案

使用 `System.Net.FtpWebRequest` 类执行这些操作。`FtpWebRequest` 可通过指定 FTP 下载 URI 从 `WebRequest` 类的 `Create` 方法生成。在下面的示例中，以来自 *C# Cookbook* 最新版的源代码作为下载的目标。针对目标文件打开一个 `FileStream`，然后由一个 `BinaryWriter` 封装。`BinaryReader` 使用来自 `FtpWebRequest` 的响应流进行创建。然后读取流并写入目标文件直到整个文件下载完成。这一系列操作在例 9-14 中的 `FtpDownloadAsync` 方法中展示。

例 9-14：使用 `System.Net.FtpWebRequest` 类

```
public static async Task FtpDownloadAsync(Uri ftpSite, string targetPath)
{
    try
    {
        FtpWebRequest request =
            (FtpWebRequest)WebRequest.Create(
            ftpSite);

        request.Credentials = new NetworkCredential("anonymous",
            "authors@oreilly.com");
        using (FtpWebResponse response =
            (FtpWebResponse)await request.GetResponseAsync())
        {
            Stream data = response.GetResponseStream();
            File.Delete(targetPath);
            Console.WriteLine(
                $"Downloading {ftpSite.AbsoluteUri} to {targetPath}...");

            byte[] byteBuffer = new byte[4096];
            using (FileStream output = new FileStream(targetPath, FileMode.CreateNew,
                FileAccess.ReadWrite,FileShare.ReadWrite, 4096, useAsync: true))
            {
                int bytesRead = 0;
                do
                {
                    bytesRead = await data.ReadAsync(byteBuffer, 0,
                        byteBuffer.Length);
                    if (bytesRead > 0)
                        await output.WriteAsync(byteBuffer, 0, bytesRead);
                }
                while (bytesRead > 0);
            }
            Console.WriteLine($"Downloaded {ftpSite.AbsoluteUri} to {targetPath}");
        }
    }
```

```
        catch (WebException e)
        {
            Console.WriteLine(
                $"Failed to download {ftpSite.AbsoluteUri} to {targetPath}");
            Console.WriteLine(e);
        }
    }
```

以下是调用 FtpDownloadAsync 的一个示例。

```
Uri downloadFtpSite =
   new Uri("ftp://ftp.oreilly.com/pub/examples/csharpckbk/CSharpCookbook.zip");
string targetPath = "CSharpCookbook.zip";
await NetworkingAndWeb.FtpDownloadAsync(downloadFtpSite, targetPath);
```

要上传一个文件，可使用 FtpWebRequest 以使用 GetRequestStream 获得请求上的一个流，并使用它上传文件。一旦文件被打开并写入请求流，就可以通过调用 GetResponse 执行请求并检查 StatusDescription 属性以获得操作的结果。这展示在如下的 FtpUploadAsync 方法中。

```
public static async Task FtpUploadAsync(Uri ftpSite, string uploadFile)
{
    Console.WriteLine($"Uploading {uploadFile} to {ftpSite.AbsoluteUri}...");
    try
    {
        FileInfo fileInfo = new FileInfo(uploadFile);
        FtpWebRequest request =
            (FtpWebRequest)WebRequest.Create(
            ftpSite);
        request.Method = WebRequestMethods.Ftp.UploadFile;
        //如果用于文本文件并需要跨操作系统平台
        //你也许会想将此值设置为false,以避免行结束符问题
        request.UseBinary = true;
        request.ContentLength = fileInfo.Length;
        request.Credentials = new NetworkCredential("anonymous",
            "authors@oreilly.com");
        byte[] byteBuffer = new byte[4096];
        using (Stream requestStream = await request.GetRequestStreamAsync())
        {
            using (FileStream fileStream =
                new FileStream(uploadFile, FileMode.Open, FileAccess.Read,
                FileShare.Read, 4096, useAsync: true))
            {
                int bytesRead = 0;
                do
                {
                    bytesRead = await fileStream.ReadAsync(byteBuffer, 0,
                                    byteBuffer.Length);
                    if (bytesRead > 0)
                        await requestStream.WriteAsync(byteBuffer, 0, bytesRead);
                }
                while (bytesRead > 0);
            }
        }
```

```
            using (FtpWebResponse response =
                  (FtpWebResponse) await request.GetResponseAsync())
            {
                Console.WriteLine(response.StatusDescription);
            }
            Console.WriteLine($"Uploaded {uploadFile} to {ftpSite.AbsoluteUri}...");
        }
        catch (WebException e)
        {
            Console.WriteLine(
                $"Failed to upload {uploadFile} to {ftpSite.AbsoluteUri}.");
            Console.WriteLine(((FtpWebResponse)e.Response).StatusDescription);
            Console.WriteLine(e);
        }
    }
```

以下是调用 `FtpUploadAsync` 方法的一个示例。

```
    string uploadFile = "SampleClassLibrary.dll";
    Uri uploadFtpSite =
        new Uri($"ftp://localhost/{uploadFile}");
    await NetworkingAndWeb.FtpUploadAsync(uploadFtpSite, uploadFile);
```

9.18.3 讨论

文件传输协议（FTP）在 RFC 959 中定义，是在互联网上分发文件的主要方式。用于 FTP 的端口号是 21。幸运的是，你不必为了使用 FTP 而真正了解很多 FTP 如何工作的信息。这会有助于应用程序自动下载来自专用 FTP 站点的信息或者提供自动上传功能。

9.18.4 参考

MSDN 文档中的“`FtpWebRequest` 类”“`FtpWebResponse` 类”“`WebRequest` 类”和“`WebResponse` 类”主题。

第10章

XML

10.0 简介

可扩展标记语言（extensible markup language，XML）是一种以结构化格式表示数据的简单、灵活、可移植的方法。XML 可用于许多方面，既可充当基于 Web 的消息协议的基础（例如 SOAP），也能作为一种存储配置数据的流行方法（例如 .NET Framework 中的 web.config、machine.config 或 security.config 文件）。Microsoft 认识到了 XML 对开发人员的用处，并且在给开发人员取舍选择方面做了良好的工作。有时候你希望以类似只读游标的方式简单地在 XML 文档中查找某个值；有些时候，你需要能够随机访问文档的各个部分；还有些时候，能够以声明方式查询并处理 XML 是很方便的。Microsoft 提供了诸如用于轻型访问的 `XmlReader` 和 `XmlWriter` 类以及用于完整文档对象模式（DOM）处理支持的 `XmlDocument` 类。为了支持以声明方式查询 XML 文档或者构建 XML，C# 以 `XElement` 和 `XDocument` 类的形式提供了 LINQ to XML（也被称为 XLINQ）。

你可能或多或少会在 .NET 中处理 XML。本章探索了 XML 和基于 XML 的技术（例如 XPath 和 XSLT）有何作用，并且展示了如何通过 LINQ to XML 来使用或者在某些时候代替这些技术。本章探讨的主题还包括 XML 验证和从 XML 到 HTML 的转换。

10.1 以文档顺序读取和访问XML数据

10.1.1 问题

你需要读取一个 XML 文档中的所有元素并获得关于每个元素的信息，例如它的名称和属性。

10.1.2　解决方案

创建一个 XmlReader 并使用它的 Read 方法去处理文档，如例 10-1 所示。

例 10-1：读取一个 XML 文档

```
public static void AccessXml()
{
    // 构建XML的新LINQ to XML语法
    XDocument xDoc = new XDocument(
                        new XDeclaration("1.0", "UTF-8", "yes"),
                        new XComment("My sample XML"),
                        new XProcessingInstruction("myProcessingInstruction",
                            "value"),
                        new XElement("Root",
                            new XElement("Node1",
                                new XAttribute("nodeId", "1"), "FirstNode"),
                            new XElement("Node2",
                                new XAttribute("nodeId", "2"), "SecondNode"),
                            new XElement("Node3",
                                new XAttribute("nodeId", "1"), "ThirdNode")
                        )
                        );

    // 将XML输出到控制台
    Console.WriteLine(xDoc.ToString());

    // 从XDocument中创建一个XmlReader
    XmlReader reader = xDoc.CreateReader();
    reader.Settings.CheckCharacters = true;
    int level = 0;
    while (reader.Read())
    {
        switch (reader.NodeType)
        {
            case XmlNodeType.CDATA:
                Display(level, $"CDATA: {reader.Value}");
                break;
            case XmlNodeType.Comment:
                Display(level, $"COMMENT: {reader.Value}");
                break;
            case XmlNodeType.DocumentType:
                Display(level, $"DOCTYPE: {reader.Name}={reader.Value}");
                break;
            case XmlNodeType.Element:
                Display(level, $"ELEMENT: {reader.Name}");
                level++;
                while (reader.MoveToNextAttribute())
                {
                    Display(level, $"ATTRIBUTE: {reader.Name}='{reader.Value}'");
                }
                break;
            case XmlNodeType.EndElement:
                level--;
                break;
```

```
                case XmlNodeType.EntityReference:
                    Display(level, $"ENTITY: {reader.Name}", reader.Name);
                    break;
                case XmlNodeType.ProcessingInstruction:
                    Display(level, $"INSTRUCTION: {reader.Name}={reader.Value}");
                    break;
                case XmlNodeType.Text:
                    Display(level, $"TEXT: {reader.Value}");
                    break;
                case XmlNodeType.XmlDeclaration:
                    Display(level, $"DECLARATION: {reader.Name}={reader.Value}");
                    break;
            }
        }
    }

    private static void Display(int indentLevel, string format, params object[] args)
    {
        for (int i = 0; i < indentLevel; i++)
            Console.Write(" ");
        Console.WriteLine(format, args);
    }
```

下面的代码以层次格式转储 XML 文档。

```
<!--My sample XML-->
<?myProcessingInstruction value?>
<Root>
  <Node1 nodeId="1">FirstNode</Node1>
  <Node2 nodeId="2">SecondNode</Node2>
  <Node3 nodeId="1">ThirdNode</Node3>
</Root>
COMMENT: My sample XML
INSTRUCTION: myProcessingInstruction=value
ELEMENT: Root
 ELEMENT: Node1
  ATTRIBUTE: nodeId='1'
  TEXT: FirstNode
 ELEMENT: Node2
  ATTRIBUTE: nodeId='2'
  TEXT: SecondNode
 ELEMENT: Node3
  ATTRIBUTE: nodeId='1'
  TEXT: ThirdNode
```

10.1.3 讨论

读取已有的 XML 并识别不同的节点类型是开发人员在处理 XML 时需要执行的基础行为。解决方案中的代码从一个声明式的结构化 XML 文档中创建了一个 `XmlReader`，然后在节点上迭代时重新创建格式化的 XML 以便向控制台窗口输出。

10.1.2 节展示了如何通过使用一个 `XDocument` 创建一个 XML 文档，并使用诸如 `XElement`、`XAttribute` 和 `XComment` 之类的各种 LINQ to XML 类构成内联 XML。

```
XDocument xDoc = new XDocument(
                    new XDeclaration("1.0", "UTF-8", "yes"),
                    new XComment("My sample XML"),
                    new XProcessingInstruction("myProcessingInstruction",
                        "value"),
                    new XElement("Root",
                        new XElement("Node1",
                            new XAttribute("nodeId", "1"), "FirstNode"),
                        new XElement("Node2",
                            new XAttribute("nodeId", "2"), "SecondNode"),
                        new XElement("Node3",
                            new XAttribute("nodeId", "1"), "ThirdNode")
                    )
                    );
```

建立 XDocument 之后，你需要对 XmlReader 进行设置，这些设置位于通过 XmlReader.Settings 属性访问的一个 XmlReaderSettings 对象。这些设置告诉 XmlReader 检查 XML 片段中的任意非法字符。

```
// 从XDocument中创建一个XmlReader
XmlReader reader = xDoc.CreateReader();
reader.Settings.CheckCharacters = true;
```

while 循环通过每次读取一个节点并检查读取器当前节点的 NodeType 属性在 XML 上进行迭代，以确定 XML 节点所属的类型。

```
while (reader.Read())
{
    switch (reader.NodeType)
    {
```

NodeType 属性是一个 XmlNodeType 枚举值，指定了可能存在的 XML 节点类型。XmlNodeType 枚举值如表 10-1 所示。

表10-1：XmlNodeType枚举值

名　称	描　述
Attribute	一个元素的属性节点
CDATA	一个标记，用于要转义的、通常被视作标记的文本段
Comment	XML 中的注释：<!-- my comment -->
Document	XML 文档树的根
DocumentFragment	文档片段节点
DocumentType	文档类型声明
Element	一个元素标签：<myelement>
EndElement	一个结束元素标签：</myelement>
EndEntity	调用 ResolveEntity 之后在实体末尾返回
Entity	实体声明
EntityReference	一个对实体的引用
None	如果 XmlReader 上的 Read 尚未被调用则返回这个节点
Notation	DTD（文档类型定义）中的一个符号

（续）

名　　称	描　　述
ProcessingInstruction	处理指令：<?pi myProcessingInstruction?>
SignificantWhitespace	当使用混合内容模式或者空白被保留时的空白
Text	节点的文本内容
Whitespace	标记实体之间的空白
XmlDeclaration	文档中第一个节点，不能拥有子节点：<?xml version='1.0'?>

10.1.4　参考

MSDN 文档中的“XmlReader 类”“XmlNodeType 枚举”和“XDocument 类”主题。

10.2　查询XML文档的内容

10.2.1　问题

你有一个很大并且很复杂的 XML 文档，需要从中找出各种各样的信息，例如某一特定元素内包含的具有特定属性设置的所有内容。你希望查询 XML 结构但不想在 XML 文档的所有节点上手动进行迭代并搜索某一特定项。

10.2.2　解决方案

使用新的 LINQ to XML API 来查询 XML 文档中感兴趣的项。LINQ 允许你根据元素和属性值选择元素，排序结果，并且返回一个基于 IEnumerable 的结果数据集合，如例 10-2 所示。

例 10-2：使用 LINQ 查询一个 XML 文档

```
private static XDocument GetAClue() => new XDocument(
                    new XDeclaration("1.0", "UTF-8", "yes"),
                    new XElement("Clue",
                        new XElement("Participant",
                            new XAttribute("type", "Perpetrator"), "Professor Plum"),
                        new XElement("Participant",
                            new XAttribute("type", "Witness"), "Colonel Mustard"),
                        new XElement("Participant",
                            new XAttribute("type", "Witness"), "Mrs. White"),
                        new XElement("Participant",
                            new XAttribute("type", "Witness"), "Mrs. Peacock"),
                        new XElement("Participant",
                            new XAttribute("type", "Witness"), "Mr. Green"),
                        new XElement("Participant",
                            new XAttribute("type", "Witness"), "Miss Scarlet"),
                        new XElement("Participant",
                            new XAttribute("type", "Victim"), "Mr. Boddy")
                    ));
```

要注意，当使用 LINQ 在 GetAClue 方法中构建该 XML 片段时，XML 的结构与代码的结构是非常相似的。

```
public static void QueryXml()
{
    XDocument xDoc = GetAClue();

    // 建立查询,查找已婚女性参与者中的证人
    var query = from p in xDoc.Root.Elements("Participant")
                where p.Attribute("type").Value == "Witness" &&
                    p.Value.Contains("Mrs.")
                orderby p.Value
                select p.Value;

    // 输出查找到的节点(本例中为Mrs. Peacock和Mrs. White,
    // 因为结果被排序了)
    foreach (string s in query)
    {
        Console.WriteLine(s);
    }
}
```

LINQ to XML 示例输出了下列内容。

```
Mrs. Peacock
Mrs. White
```

如果不使用 LINQ 来查询一个 XML 文档，你还可以使用 XPath。在 .NET 中，这意味着使用 System.Xml.XPath 命名空间以及诸如 XPathDocument、XPathNavigator 和 XPathNodeIterator 的类。LINQ to XML 还支持通过 XElement.XPathSelectElements 方法在一个查询使用 XPath 来识别数据项。

在例 10-3 中，你要使用这些类从一个包含桌游 *Clue*（在北美之外也被称为 *Cluedo*）成员及其各种角色的 XML 文档中选择节点。你希望能够选择作为犯罪证人的已婚女性参与者。为此，要传递一个 XPath 表达式去查询该 XML 数据集，如例 10-3 所示。

例 10-3：使用 XPath 查询一个 XML 文档

```
public static void QueryXML()
{
    XDocument xDoc = GetAClue();

    using (StringReader reader = new StringReader(xDoc.ToString()))
    {
        // 使用StringReader实例化一个XPathDocument
        XPathDocument xpathDoc = new XPathDocument(reader);

        // 获得导航器
        XPathNavigator xpathNav = xpathDoc.CreateNavigator();

        // 建立查询查找已婚的女性参与者中的证人
        string xpathQuery =
            "/Clue/Participant[attribute::type='Witness'][contains(text(),'Mrs.')]";
        XPathExpression xpathExpr = xpathNav.Compile(xpathQuery);
```

```
            // 从已编译的表达式中获得节点集
            XPathNodeIterator xpathIter = xpathNav.Select(xpathExpr);

            // 输出查找到的节点(本例中为Mrs. White和Mrs. Peacock)
            while (xpathIter.MoveNext())
            {
                Console.WriteLine(xpathIter.Current.Value);
            }
        }
    }
```

XPath 示例会输出下列内容。

```
Mrs. White
Mrs. Peacock
```

10.2.3 讨论

当使用 LINQ 时，查询支持在 C# 中是一等公民。与 XPath 相比，LINQ to XML 为大多数开发人员编写查询带来了更直观的语法，因此是一种很受欢迎的语言新增功能。如果你需要应对的是可扩展处理 XML 的系统，那么 XPath 将会是一种值得拥有的有用工具。但是，在许多情况下，你知道自己的要求，只是不知道 XPath 的语法。甚至对于那些拥有很少 SQL 经验的开发人员，在 C# 中查询也会变得更加容易。

本范例涉及的 XML 如下所示。

```
<?xml version='1.0'?>
<Clue>
  <Participant type="Perpetrator">Professor Plum</Participant>
  <Participant type="Witness">Colonel Mustard</Participant>
  <Participant type="Witness">Mrs. White</Participant>
  <Participant type="Witness">Mrs. Peacock</Participant>
  <Participant type="Witness">Mr. Green</Participant>
  <Participant type="Witness">Miss Scarlet</Participant>
  <Participant type="Victim">Mr. Boddy</Participant>
</Clue>
```

这个查询的意思是“挑选 Participant 是一个证人并且其称呼为 Mrs. 的所有 Participant 元素”。

```
// 建立查询,查找已婚的女性参与者中的证人
var query = from p in xDoc.Root.Elements("Participant")
            where p.Attribute("type").Value == "Witness" &&
                p.Value.Contains("Mrs.")
            orderby p.Value
            select p.Value;
```

将这个查询与使用 XPath 语法的相同查询相比较。

```
// 建立查询,查找已婚的女性参与者中的证人
string xpathQuery =
  "/Clue/Participant[attribute::type='Witness'][contains(text(),'Mrs.')]";
```

两种执行查询的方法都有价值，但要考虑的问题是后续的开发人员如何能够容易地理解你所编写的代码。很容易就会破坏没能很好理解的代码。

> 一般而言，理解 SQL 的开发人员比理解 XPath 的多，即使有了现在可用的所有 Web 服务也是如此。这一点可能与你自己的经历有所不同，如果你做了大量跨平台的工作，那么尤其如此；但关键在于不仅仅将 LINQ 理解为另一种语法，而是作为一种令你的代码让更多开发人员易于阅读的方法。代码很少被某个人拥有，即使在短期内也是如此。因此，为什么不让那些接替你的人轻松点呢？毕竟有一天你也会站在接替别人的位置上。让我们进一步分解这两个查询。

LINQ 查询使用了 C# 中的以下这些关键字。

- `var` 指示编译器期望根据结果集推断一个类型。
- `from` 也称为生成器，为查询提供一个可进行操作的数据源以及允许访问单个元素的范围变量。
- `where` 允许将一个布尔条件应用于数据源的每个元素上，以确定它是否应当包含在结果数据集合中。
- `orderby` 根据元素的数目和每个元素的升序或降序指示器对结果集进行排序。针对排序的多个级别可指定多个标准。
- `select` 表示所有条件求值后将被返回的值序列。这也称为值的投影。

这意味着我们的语法可以归结为以下几点。

- `from p in xDoc.Root.Elements("Participant")` 说明“获得所有位于 `Clue` 根级节点之下的 `Participant`”。
- `where p.Attribute("type").Value == "Witness"` 说明“只选择带有名为 `type` 的属性且属性值为 `Witness` 的 `Participant`”。
- `&& p.Value.Contains("Mrs.")` 说明“只选择其值包含 `Mrs.` 的 `Participant`”。
- `orderby p.Value` 说明“按名称以升序排序参加者”。
- `select p.Value` 说明“选择已满足之前所有标准的 `Participant` 元素的值”。

XPath 语法执行相同的功能。

- `/Clue/Participants` 说明“获得所有位于 `Clue` 根级节点之下的 `Participant`”。
- `Participant[attribute::type='Witness']` 说明“只选择带有名为 `type` 的属性且属性值为 `Witness` 的 `Participant`”。
- `Participant[contains(text(),'Mrs.')]` 说明“只选择其值包含 `Mrs.` 的 `Participant`”。

将它们放在一起，你就可以在两种情况下获得作为证人的所有已婚女性参与者，并且使用 LINQ 对结果进行排序。

10.2.4　参考

MSDN 文档中的“查询表达式”“`XElement` 类”和“XPath，读取 XML”主题。

10.3 验证XML

10.3.1 问题

你正在接收一个由其他来源创建的 XML 文档，并且希望验证它是否遵循某一特定架构。这一模式可以是 XML 架构（XSD 或 XML-XDR）的形式，或者是希望能灵活使用文档类型定义（DTD）来验证 XML。

10.3.2 解决方案

使用 `XDocument.Validate` 方法和 `XmlReader.Settings` 属性来验证 XML 文档与其他描述符文档（如 XSD、DTD 或者 XDR），如例 10-4 所示。作为测试的一部分来验证从你的软件中生成的 XML，将会使你从稍后整合其他系统（或者系统的组件）时的 bug 中解放出来，并且强烈鼓励这样做！

例 10-4：验证 XML

```
public static void ValidateXml()
{
    // 打开bookbad.xml文件
    XDocument book = XDocument.Load(@"..\..\BookBad.xml");
    // 使用book.xsd创建XSD架构集合
    XmlSchemaSet schemas = new XmlSchemaSet();
    schemas.Add(null,@"..\..\Book.xsd");
    // 装配处理程序以获得任意验证错误通知
    book.Validate(schemas, settings_ValidationEventHandler);

    // 创建一个读取器以读取文件,从而触发验证
    XmlReader reader = book.CreateReader();
    // 同时报告错误和警告
    reader.Settings.ValidationFlags =
        XmlSchemaValidationFlags.ReportValidationWarnings;
    // 使用XML架构
    reader.Settings.ValidationType = ValidationType.Schema;
    // 读取XML
    while (reader.Read())
    {
        if (reader.NodeType == XmlNodeType.Element)
        {
            Console.Write($"<{reader.Name}");
            while (reader.MoveToNextAttribute())
            {
                Console.Write($"{reader.Name}='{reader.Value}'");
            }
            Console.Write(">");
        }
        else if (reader.NodeType == XmlNodeType.Text)
        {
            Console.Write(reader.Value);
        }
        else if (reader.NodeType == XmlNodeType.EndElement)
```

```
            {
                Console.WriteLine($"</{reader.Name}>");
            }
        }
    }

    private static void settings_ValidationEventHandler(object sender,
        ValidationEventArgs e)
    {
        Console.WriteLine($"Validation Error Message: {e.Message}");
        Console.WriteLine($"Validation Error Severity: {e.Severity}");
        Console.WriteLine($"Validation Error Line Number: {e.Exception?.LineNumber}");
        Console.WriteLine(
            $"Validation Error Line Position: {e.Exception?.LinePosition}");
        Console.WriteLine($"Validation Error Source: {e.Exception?.Source}");
        Console.WriteLine($"Validation Error Source Schema: " +
                          $"{{e.Exception?.SourceSchemaObject}");
        Console.WriteLine($"Validation Error Source Uri: {e.Exception?.SourceUri}");
        Console.WriteLine($"Validation Error thrown from: {e.Exception?.TargetSite}");
        Console.WriteLine($"Validation Error callstack: {e.Exception?.StackTrace}");
    }
```

10.3.3 讨论

10.3.2 节举例说明了如何使用 XDocument 和 XmlReader 去验证 book.xml 文档是否符合 book.xsd XSD 定义文件。DTD 是指定一个 XML 文档结构的最初方法，但是由于 XSD 在 2001 年到达了 W3C 推荐状态，使用 XSD 已经变得更加常见。XDR 是由 Microsoft 提供的 XSD 的前身。虽然它可能会在现存系统中遇到，但不应该用于新的开发。

要做的第一件事是创建一个 XmlSchemaSet 以持有你的 XSD 文件（book.xsd）并且调用 Add 方法向 XmlSchemaSet 添加该 XSD。使用 XmlSchemaSet 和用于验证事件的处理程序方法调用 XDocument 上的 Validate 方法。现在，验证已基本设置，更多的选项可在从 XDocument 中创建的 XmlReader 上设置。XmlReaderSettings 上的 ValidationFlags 属性允许注册检验中的警告、在验证过程中处理标识约束、处理内联架构、以及允许那些可能在模式中未定义的属性。

```
// 使用book.xsd创建XSD架构集合
XmlSchemaSet schemas = new XmlSchemaSet();
schemas.Add(null,@"..\..\Book.xsd");
// 装配处理程序以获得任意验证错误通知
book.Validate(schemas, settings_ValidationEventHandler);

// 创建一个读取器以读取文件,从而触发验证
XmlReader reader = book.CreateReader();
// 同时报告错误和警告
reader.Settings.ValidationFlags =
    XmlSchemaValidationFlags.ReportValidationWarnings;
// 使用XML架构
reader.Settings.ValidationType = ValidationType.Schema;
```

> 要执行 DTD 验证，可使用一个 DTD 和 `ValidationType.DTD`，要执行 XDR 验证，使用一个 XDR 架构和 `ValidationType.XDR`。

随后当一个验证错误发生时，`settings_ValidationEventHandler` 函数检查传入的 `ValidationEventArgs` 对象并且将相关信息写入控制台。

```
private static void settings_ValidationEventHandler(object sender,
    ValidationEventArgs e)
{
    Console.WriteLine($"Validation Error Message: {e.Message}");
    Console.WriteLine($"Validation Error Severity: {e.Severity}");
    Console.WriteLine(
        $"Validation Error Line Number: {e.Exception?.LineNumber}");
    Console.WriteLine(
        $"Validation Error Line Position: {e.Exception?.LinePosition}");
    Console.WriteLine($"Validation Error Source: {e.Exception?.Source}");
    Console.WriteLine($"Validation Error Source Schema: " +
                      $"{{e.Exception?.SourceSchemaObject}");
    Console.WriteLine($"Validation Error Source Uri: {e.Exception?.SourceUri}");
    Console.WriteLine(
        $"Validation Error thrown from: {e.Exception?.TargetSite}");
    Console.WriteLine($"Validation Error callstack: {e.Exception?.StackTrace}");
}
```

然后继续在 XML 文档上进行工作并且写出元素和属性。

```
while (readerOld.Read())
{
    if (readerOld.NodeType == XmlNodeType.Element)
    {
        Console.Write($"<{readerOld.Name}");
        while (reader.MoveToNextAttribute())
        {
            Console.Write($"{readerOld.Name}='{readerOld.Value}'");
        }
        Console.Write(">");
    }
    else if (readerOld.NodeType == XmlNodeType.Text)
    {
        Console.Write(reader.Value);
    }
    else if (readerOld.NodeType == XmlNodeType.EndElement)
    {
        Console.WriteLine($"</{readerOld.Name}>");
    }
}
```

BookBad.xml 文件包含下列内容。

```
<?xml version="1.0" encoding="utf-8"?>
<Book xmlns="http://tempuri.org/Book.xsd" name="C# Cookbook">
```

```
    <Chapter>File System IO</Chapter>
    <Chapter>Security</Chapter>
    <Chapter>Data Structures and Algorithms</Chapter>
    <Chapter>Reflection</Chapter>
    <Chapter>Threading and Synchronization</Chapter>
    <Chapter>Numbers and Enumerations</Chapter>
    <BadElement>I don't belong here</BadElement>
    <Chapter>Strings and Characters</Chapter>
    <Chapter>Classes And Structures</Chapter>
    <Chapter>Collections</Chapter>
    <Chapter>XML</Chapter>
    <Chapter>Delegates, Events, and Anonymous Methods</Chapter>
    <Chapter>Diagnostics</Chapter>
    <Chapter>Toolbox</Chapter>
    <Chapter>Unsafe Code</Chapter>
    <Chapter>Regular Expressions</Chapter>
    <Chapter>Generics</Chapter>
    <Chapter>Iterators and Partial Types</Chapter>
    <Chapter>Exception Handling</Chapter>
    <Chapter>Web</Chapter>
    <Chapter>Networking</Chapter>
</Book>
```

book.xsd 文件包含下列内容。

```
<?xml version="1.0" ?>
<xs:schema id="NewDataSet" targetNamespace="http://tempuri.org/Book.xsd"
xmlns:mstns="http://tempuri.org/Book.xsd"
    xmlns="http://tempuri.org/Book.xsd"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
    attributeFormDefault="qualified" elementFormDefault="qualified">
  <xs:element name="Book">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Chapter" nillable="true"
                    minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:simpleContent
msdata:ColumnName="Chapter_Text" msdata:Ordinal="0">
              <xs:extension base="xs:string">
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="name" form="unqualified" type="xs:string"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

当它运行时，会生成下列输出，显示发生在 `BadElement` 上的验证失败。

```
Validation Error Message: The element 'Book' in namespace 'http://tempuri.org/Bo
ok.xsd' has invalid child element 'BadElement' in namespace 'http://tempuri.org/
```

```
Book.xsd'. List of possible elements expected: 'Chapter' in namespace 'http://te
mpuri.org/Book.xsd'.
Validation Error Severity: Error
Validation Error Line Number: 0
Validation Error Line Position: 0
Validation Error Source:
Validation Error Source Schema:
Validation Error Source Uri:
Validation Error thrown from:
Validation Error callstack:
<Book xmlns='http://tempuri.org/Book.xsd' name='C# Cookbook'><Chapter>File Syste
m IO</Chapter>
<Chapter>Security</Chapter>
<Chapter>Data Structures and Algorithms</Chapter>
<Chapter>Reflection</Chapter>
<Chapter>Threading and Synchronization</Chapter>
<Chapter>Numbers and Enumerations</Chapter>
<BadElement>I don't belong here</BadElement>
<Chapter>Strings and Characters</Chapter>
<Chapter>Classes And Structures</Chapter>
<Chapter>Collections</Chapter>
<Chapter>XML</Chapter>
<Chapter>Delegates, Events, and Anonymous Methods</Chapter>
<Chapter>Diagnostics</Chapter>
<Chapter>Toolbox</Chapter>
<Chapter>Unsafe Code</Chapter>
<Chapter>Regular Expressions</Chapter>
<Chapter>Generics</Chapter>
<Chapter>Iterators and Partial Types</Chapter>
<Chapter>Exception Handling</Chapter>
<Chapter>Web</Chapter>
<Chapter>Networking</Chapter>
</Book>
```

10.3.4　参考

MSDN 文档中的“XmlReader 类”“XmlSchemaSet 类”“ValidationType 枚举”和“XDocument 类”主题。

10.4　检测对XML文档的修改

10.4.1　问题

你需要通知一个或多个类或者组件：在 XML 文档中已经插入或移除了某个节点或者其值已改变。

10.4.2　解决方案

为了跟踪对一个活跃的 XML 文档的修改，订阅 XDocument 类发布的事件。XDocument 发布了关于一个节点何时将发生改变以及它何时发生了改变的事件，分别用于节点变化的前置

和后置状态。

例 10-5 展示了许多在与 DetectXMLChanges 方法相同的作用域中定义的事件处理程序，但它们可以很容易作为对操作在线 XML 文档感兴趣的其他类上的函数进行回调。

DetectXMLChanges 加载在方法中定义的 XML 片段，将事件处理程序与节点事件连接；添加、修改并移除节点以触发事件，然后写出结果 XML。

例 10-5：检测对 XML 文档的修改

```
public static void DetectXmlChanges()
{
    XDocument xDoc = new XDocument(
                        new XDeclaration("1.0", "UTF-8", "yes"),
                        new XComment("My sample XML"),
                        new XProcessingInstruction("myProcessingInstruction",
                            "value"),
                        new XElement("Root",
                            new XElement("Node1",
                                new XAttribute("nodeId", "1"), "FirstNode"),
                            new XElement("Node2",
                                new XAttribute("nodeId", "2"), "SecondNode"),
                            new XElement("Node3",
                                new XAttribute("nodeId", "1"), "ThirdNode"),
                            new XElement("Node4",
                                new XCData(@"<>\&'"))
                        )
                        );
    // 创建事件处理程序
    xDoc.Changing += xDoc_Changing;
    xDoc.Changed += xDoc_Changed;
    // 添加一个元素节点
    XElement element = new XElement("Node5", "Fifth Element");
    xDoc.Root.Add(element);

    // 修改第一个节点
    // doc.DocumentElement.FirstChild.InnerText = "1st Node";
    if(xDoc.Root.FirstNode.NodeType == XmlNodeType.Element)
        ((XElement)xDoc.Root.FirstNode).Value = "1st Node";

    // 移除第4个节点
    var query = from e in xDoc.Descendants()
                where e.Name.LocalName == "Node4"
                select e;
    XElement[] elements = query.ToArray<XElement>();
    foreach (XElement xelem in elements)
    {
        xelem.Remove();
    }
    // 输出新的xml
    Console.WriteLine();
    Console.WriteLine(xDoc.ToString());
    Console.WriteLine();
}
```

例 10-6 给出来自 XDocument 的事件处理程序以及格式化方法 WriteElementInfo。该方法接受一个动作字符串并获得要操作对象的名称和值。两个事件处理程序都调用该格式化方法，传递相应的动作字符串。

例 10-6：XDocument 事件处理程序和 WriteElementInfo 方法

```
private static void xDoc_Changed(object sender, XObjectChangeEventArgs e)
{
    // Add - 一个XObject已被或将要被添加到XContainer
    // Name - 一个XObject已被或将要被重命名
    // Remove - 一个XObject已被或将要被从XContainer中移除
    // Value - XObject的值已被或将要被修改;此外,一个空元素
    // (无论是从一个空标记到开始/结束标记或者相反)的序列化的修改
    // 也会触发此事件
    WriteElementInfo("changed", e.ObjectChange, (XObject)sender);
}

private static void xDoc_Changing(object sender, XObjectChangeEventArgs e)
{
    // Add - 一个XObject已被或将要被添加到XContainer
    // Name - 一个XObject已被或将要被重命名
    // Remove - 一个XObject已被或将要被从XContainer中移除
    // Value - XObject的值已被或将要被修改;此外,一个空元素
    // (无论是从一个空标记到开始/结束标记或者相反)的序列化的修改
    // 也会触发此事件
    WriteElementInfo("changing", e.ObjectChange, (XObject)sender);
}

private static void WriteElementInfo(string action, XObjectChange change,
    XObject xobj)
{
    if (xobj != null)
        Console.WriteLine($"XObject: <{xobj.NodeType.ToString()}> "+
            $"{action} {change} with value {xobj}");
    else
        Console.WriteLine("XObject: <{xobj.NodeType.ToString()}> " +
            $"{action} {change} with null value");
}
```

DetectXMLChanges 方法的结果如下所示。

```
XObject: <Element> changing Add with value <Node5>Fifth Element</Node5>
XObject: <Element> changed Add with value <Node5>Fifth Element</Node5>
XObject: <Text> changing Remove with value FirstNode
XObject: <Text> changed Remove with value FirstNode
XObject: <Text> changing Add with value 1st Node
XObject: <Text> changed Add with value 1st Node
XObject: <Element> changing Remove with value <Node4><![CDATA[<>\&']]></Node4>
XObject: <Element> changed Remove with value <Node4><![CDATA[<>\&']]></Node4>

<!--My sample XML-->
<?myProcessingInstruction value?>
<Root>
  <Node1 nodeId="1">1st Node</Node1>
  <Node2 nodeId="2">SecondNode</Node2>
```

```
  <Node3 nodeId="1">ThirdNode</Node3>
  <Node5>Fifth Element</Node5>
</Root>
```

10.4.3 讨论

XDocument 类由 XElement 类派生而来。XDocument 还可能包含一个文档类型声明（XDocumentType）、一个根元素（XDocument.Root）、注释（XComment）以及处理指令（XProcessingInstruction）。通常来说，你将会使用 XElement 构建大多数 XML 文档的类型，但如果需要指定上述项中的任何一个，请使用 XDocument。

10.4.4 参考

MSDN 文档中的“XDocument 类”和“XObjectChangeEventHandler 委托”主题。

10.5 处理XML字符串中的无效字符

10.5.1 问题

你正在创建一个 XML 字符串。在添加一个包含文本元素的标签之前，你希望检查它以确定字符串是否包含下列非法字符。

```
<
>
"
'
&
```

如果遇到这些字符中的任何一个，你希望用它们的转义形式进行替换。下面是它们的转义形式。

```
&lt; (<)
&gt; (>)
&quot; (")
&apos; (')
&amp; (&)
```

10.5.2 解决方案

根据你所使用的 XML 创建方法，完成这项工作有多种不同的方法。如果使用 XElement，那么使用 XCData 对象或者直接作为 XElement 的值添加文本将会处理正确的转义。如果使用 XmlWriter，那么 WriteCData、WriteString、WriteAttributeString、WriteValue 和 WriteElementString 方法会为你处理好它们。如果使用 XmlDocument 和 XmlElement，那么 XmlElement.InnerText 方法将会处理这些字符。

在使用 XElement 处理非法字符的第一种方法中，XCData 对象将非法字符文本封装在一个 CDATA 节中，如后面示例中 InvalidChars1 元素的生成所示。另一种使用 XElement 的方法

将文本赋给 XElement 的值，并且自动转义文本，如创建 InvalidChars2 元素时所示。

```
// 建立一个包含非法字符的字符串
string invalidChars = @"<>\&'";
XElement element = new XElement("Root",
                        new XElement("InvalidChars1",
                            new XCData(invalidChars)),
                        new XElement("InvalidChars2",invalidChars));
Console.WriteLine($"Generated XElement with Invalid Chars:\r\n{element}");
Console.WriteLine();
```

它的输出如下所示。

```
Generated XElement with Invalid Chars:
<Root>
  <InvalidChars1><![CDATA[<>\&']]></InvalidChars1>
  <InvalidChars2>&lt;&gt;\&amp;'</InvalidChars2>
</Root>
```

在使用 XmlWriter 处理非法字符的第一种方法中，WriteCData 方法将非法字符文本封装在一个 CDATA 节中，如后面示例中 InvalidChars1 元素的生成所示。另一种使用 XmlWriter 的方法使用 WriteElementString 方法，自动转义文本，如创建 InvalidChars2 元素时所示。

```
// 建立一个包含非法字符的字符串
string invalidChars = @"<>\&'";
XmlWriterSettings settings = new XmlWriterSettings();
settings.Indent = true;
using (XmlWriter writer = XmlWriter.Create(Console.Out, settings))
{
    writer.WriteStartElement("Root");
    writer.WriteStartElement("InvalidChars1");
    writer.WriteCData(invalidChars);
    writer.WriteEndElement();
    writer.WriteElementString("InvalidChars2", invalidChars);
    writer.WriteEndElement();
}
```

它的输出如下所示。

```
<?xml version="1.0" encoding="IBM437"?>
<Root>
  <InvalidChars1><![CDATA[<>\&']]></InvalidChars1>
  <InvalidChars2>&lt;&gt;\&amp;'</InvalidChars2>
</Root>
```

使用 XmlDocument 和 XmlElement 处理该问题有两种方法：第一种方法是将要添加的文本放置在一个 CDATA 节中，并将其添加到 XmlElement 的 InnerXML 属性中。

```
// 建立一个包含非法字符的字符串
string invalidChars = @"<>\&'";

// 创建第一个非法字符节点
XmlElement invalidElement1 = xmlDoc.CreateElement("InvalidChars1");

// 将非法字符封装在一个CDATA节中,并且使用InnerXML属性以赋值,
```

```
// 因为它并不会转义值,只需要传入提供的文本
invalidElement1.AppendChild(xmlDoc.CreateCDataSection(invalidChars));
```

第二种方法如下所示，通过将文本直接赋给 InnerText 属性，让 XmlElement 类为你封装数据。

```
// 建立一个包含非法字符的字符串
string invalidChars = @"<>\&'";
// 创建第二个非法字符节点
XmlElement invalidElement2 = xmlDoc.CreateElement("InvalidChars2");

// 直接使用InnerText属性添加非法字符以赋值,
// 因为它会自动转义值
invalidElement2.InnerText = invalidChars;

// 将元素追加到根节点
root.AppendChild(invalidElement2);
```

在该代码中，使用 XmlElement 生成了整个 XmlDocument。

```
public static void HandleInvalidChars()
{
    // 建立一个包含非法字符的字符串
    string invalidChars = @"<>\&'";
    XElement element = new XElement("Root",
                            new XElement("InvalidChars1",
                                new XCData(invalidChars)),
                            new XElement("InvalidChars2",invalidChars));
    Console.WriteLine($"Generated XElement with Invalid Chars:\r\n{element}");
    Console.WriteLine();

    XmlWriterSettings settings = new XmlWriterSettings();
    settings.Indent = true;
    using (XmlWriter writer = XmlWriter.Create(Console.Out, settings))
    {
        writer.WriteStartElement("Root");
        writer.WriteStartElement("InvalidChars1");
        writer.WriteCData(invalidChars);
        writer.WriteEndElement();
        writer.WriteElementString("InvalidChars2", invalidChars);
        writer.WriteEndElement();
    }
    Console.WriteLine();

    XmlDocument xmlDoc = new XmlDocument();
    // 创建文档的根节点
    XmlElement root = xmlDoc.CreateElement("Root");
    xmlDoc.AppendChild(root);

    // 创建第一个非法字符节点
    XmlElement invalidElement1 = xmlDoc.CreateElement("InvalidChars1");
    // 将非法字符封装在一个CDATA节中,并且使用InnerXML属性以赋值,
    // 因为它并不会转义值,只需要传入提供的文本
    invalidElement1.AppendChild(xmlDoc.CreateCDataSection(invalidChars));
    // 将元素追加到根节点
```

```
        root.AppendChild(invalidElement1);

        // 创建第二个非法字符节点
        XmlElement invalidElement2 = xmlDoc.CreateElement("InvalidChars2");
        // 直接使用InnerText属性添加非法字符以赋值，
        // 因为它会自动转义值
        invalidElement2.InnerText = invalidChars;
        // 将元素追加到根节点
        root.AppendChild(invalidElement2);

        Console.WriteLine($"Generated XML with Invalid Chars:\r\n{xmlDoc.OuterXml}");
        Console.WriteLine();
    }
```

这一过程所生成的 XML（以及控制台的输出）如下所示。

```
Generated XML with Invalid Chars:
<Root><InvalidChars1><![CDATA[<>\&']]></InvalidChars1><InvalidChars2>&lt;&gt;\&a
mp;'</InvalidChars2></Root>
```

10.5.3 讨论

为便于输入，CDATA 节点允许将文本片段中的项表示为字符数据，而不是作为转义的 XML。一般而言，这些字符需要处于其转义格式（例如，对于 < 是 <），但是 CDATA 节允许你将其作为常规文本键入。

当 CDATA 标签与 XmlElement 类的 InnerXml 一起使用时，你可以提交通常需要被首先转义的字符。XmlElement 类还拥有一个 InnerText 属性，它将自动转义所有被赋值的字符串中发现的标记。这使得你可以添加这些字符而无需担心它们。

10.5.4 参考

MSDN 文档中的“XElement 类”“XCData 类”“XmlDocument 类”“XmlWriter 类”“XmlElement 类”和“CDATA 节”主题。

10.6 转换XML

10.6.1 问题

你有一个原始的 XML 文档，需要将它转换为一种更易阅读的格式。例如，你拥有存储为 XML 文档的个人数据，需要将它显示在一个网页上或者放置在以逗号分隔的文本文件中用于原有系统集成。不幸的是，并非所有人都希望整天对大量 XML 进行分类，他们更希望以一种格式化的列表形式或者在一个定义好列和行的表格中阅读数据。你需要一个方法将 XML 数据转换为一种更易读的形式以及以逗号分隔的格式。

10.6.2 解决方案

本节的解决方案使用 LINQ to XML 在 C# 中执行转换。在示例代码中，转换了存储在

Personnel.xml 中来自虚拟公司的一些个人数据。首先将数据转换为 HTML，然后转换为以逗号分隔的格式。

```
// LINQ方式
XElement personnelData = XElement.Load(@"..\..\Personnel.xml");
// 创建HTML
XElement personnelHtml =
    new XElement("html",
        new XElement("head"),
        new XElement("body",
            new XAttribute("title","Personnel"),
            new XElement("p",
                new XElement("table",
                    new XAttribute("border","1"),
                    new XElement("thead",
                        new XElement("tr",
                            new XElement("td","Employee Name"),
                            new XElement("td","Employee Title"),
                            new XElement("td","Years with Company"),
                            new XElement("td","Also Known As")
                            )
                        ),
                    new XElement("tbody",
                        from p in personnelData.Elements("Employee")
                        select new XElement("tr",
                            new XElement("td", p.Attribute("name").Value),
                            new XElement("td", p.Attribute("title").Value),
                            new XElement("td",
                                p.Attribute("companyYears").Value),
                            new XElement("td", p.Attribute("nickname").Value)
                            )
                        )
                    )
                )
            )
        );

personnelHtml.Save(@"..\..\Personnel_LINQ.html");

var queryCSV = from p in personnelData.Elements("Employee")
                orderby p.Attribute("name").Value descending
                select p;
StringBuilder sb = new StringBuilder();
foreach(XElement e in queryCSV)
{
    sb.AppendFormat($"{EscapeAttributeForCSV(e, "name")}," +
        $"{EscapeAttributeForCSV(e, "title")}," +
        $"{EscapeAttributeForCSV(e, "companyYears")}," +
        $"{EscapeAttributeForCSV(e, "nickname")}" +
        $"{Environment.NewLine}");
}
using(StreamWriter writer = File.CreateText(@"..\..\Personnel_LINQ.csv"))
{
    writer.Write(sb.ToString());
}
```

从 LINQ 到 CSV 的转换输出如下所示。

```
Rutherford,CEO,27,""BigTime""
Chas,Salesman,3,""Money""
Bob,Customer Service,1,""Happy""
Alice,Manager,12,""Business""
```

Personnel.xml 文件包含下列项。

```
<?xml version="1.0" encoding="utf-8"?>
<Personnel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <Employee name="Bob" title="Customer Service" companyYears="1"
        nickname="&quot;Happy&quot;"/>
    <Employee name="Alice" title="Manager" companyYears="12"
        nickname="&quot;Business&quot;"/>
    <Employee name="Chas" title="Salesman" companyYears="3"
        nickname="&quot;Money&quot;"/>
    <Employee name="Rutherford" title="CEO" companyYears="27"
        nickname="&quot;BigTime&quot;"/>
</Personnel>
```

你可能想知道为什么昵称属性值在 CSV 输出中有额外的双引号。这是为了支持 RFC 4180“常见格式和 MIME 类型为 CSV 文件”中所说的“如果双引号被用于括起字段，那么字段内出现双引号时必须通过在它前面加上另一个双引号进行转义。”我们使用 `EscapeAttributeForCSV` 方法完成此操作。

```
private static string EscapeAttributeForCSV(XElement element,
    string attributeName)
{
    string attributeValue = element.Attribute(attributeName).Value;
    //RFC-4180,段落描述“如果双引号被用于括起字段,
    //那么字段内出现双引号时,
    //必须通过在它前面加上另一个双引号进行转义”
    return attributeValue.Replace("\"", "\"\"");
}
```

此方法在范例 10.8（即 10.8 节）中会进一步讨论。

我们还可以通过使用 `XslCompiledTransform` 类以使用一个 XSLT 样式表将 XML 转换为其他格式，来实现这一解决方案。首先，加载用于生成 HTML 输出的样式表，然后通过 XSLT 使用 PersonnelHTML.xsl 样式表执行到 HTML 的转换。之后，使用 PersonnelCSV.xsl 样式表将数据转换为以逗号分隔的格式。

```
// 使用默认凭据创建一个解析器
XmlUrlResolver resolver = new XmlUrlResolver();
resolver.Credentials = System.Net.CredentialCache.DefaultCredentials;

// 将personnel.xml文件转换为html
XslCompiledTransform transform = new XslCompiledTransform();
XsltSettings settings = new XsltSettings();
// 为安全原因禁用这两个属性(默认false)
settings.EnableDocumentFunction = false;
settings.EnableScript = false;
// 加载样式表
```

```
transform.Load(@"..\..\PersonnelHTML.xsl",settings,resolver);
// 执行转换
transform.Transform(@"..\..\Personnel.xml",@"..\..\Personnel.html");
```

PersonnelHTML.xsl 样式表如下所示。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xsl:template match="/">
    <html>
      <head />
      <body title="Personnel">
        <xsl:for-each select="Personnel">
          <p>
            <xsl:for-each select="Employee">
              <xsl:if test="position()=1">
                <table border="1">
                  <thead>
                    <tr>
                      <td>Employee Name</td>
                      <td>Employee Title</td>
                      <td>Years with Company</td>
                      <td>Also Known As</td>
                    </tr>
                  </thead>
                  <tbody>
                    <xsl:for-each select="../Employee">
                      <tr>
                        <td>
                          <xsl:for-each select="@name">
                            <xsl:value-of select="." />
                          </xsl:for-each>
                        </td>
                        <td>
                          <xsl:for-each select="@title">
                            <xsl:value-of select="." />
                          </xsl:for-each>
                        </td>
                        <td>
                          <xsl:for-each select="@companyYears">
                            <xsl:value-of select="." />
                          </xsl:for-each>
                        </td>
                        <td>
                          <xsl:for-each select="@nickname">
                            <xsl:value-of select="." />
                          </xsl:for-each>
                        </td>
                      </tr>
                    </xsl:for-each>
                  </tbody>
                </table>
              </xsl:if>
            </xsl:for-each>
```

```
          </p>
        </xsl:for-each>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

要生成如图 10-1 所示的 HTML 屏幕输出，可使用 PersonnelHTML.xsl 样式表和 Personnel.xml 文件。

Employee Name	Employee Title	Years with Company	Also Known As
Bob	Customer Service	1	"Happy"
Alice	Manager	12	"Business"
Chas	Salesman	3	"Money"
Rutherford	CEO	27	"BigTime"

图 10-1：由 Personnel.xml 生成的 Personnel HTML 表格

下面是 LINQ 转换生成的 HTML 源代码。

```
<?xml version="1.0" encoding="utf-8"?>
<html>
  <head />
  <body title="Personnel">
    <p>
      <table border="1">
        <thead>
          <tr>
            <td>Employee Name</td>
            <td>Employee Title</td>
            <td>Years with Company</td>
            <td>Also Known As</td>
          </tr>
        </thead>
        <tbody>
          <tr>
            <td>Bob</td>
            <td>Customer Service</td>
            <td>1</td>
            <td>"Happy"</td>
          </tr>
          <tr>
            <td>Alice</td>
            <td>Manager</td>
            <td>12</td>
            <td>"Business"</td>
          </tr>
          <tr>
            <td>Chas</td>
            <td>Salesman</td>
            <td>3</td>
            <td>"Money"</td>
```

```
            </tr>
            <tr>
              <td>Rutherford</td>
              <td>CEO</td>
              <td>27</td>
              <td>"BigTime"</td>
            </tr>
          </tbody>
        </table>
      </p>
    </body>
  </html>
```

下面是 XSLT 转换生成的 HTML 源代码。

```
  <?xml version="1.0" encoding="utf-8"?>
  <html>
    <head />
    <body title="Personnel">
      <table border="1">
        <thead>
          <tr>
            <td>Employee Name</td>
            <td>Employee Title</td>
            <td>Years with Company</td>
          </tr>
        </thead>
        <tbody>
          <tr>
            <td name="Bob" />
            <td title="Customer Service" />
            <td name="Bob" />
          </tr>
          <tr>
            <td name="Alice" />
            <td title="Manager" />
            <td name="Alice" />
          </tr>
          <tr>
            <td name="Chas" />
            <td title="Salesman" />
            <td name="Chas" />
          </tr>
          <tr>
            <td name="Rutherford" />
            <td title="CEO" />
            <td name="Rutherford" />
          </tr>
        </tbody>
      </table>
    </body>
  </html>
```

要生成以逗号分隔的输出，可使用 PersonnelCSV.xsl 和 Personnel.xml。

```
// 将personnel.xml文件转换为以逗号分隔的格式

// 加载样式表
XslCompiledTransform transformCSV = new XslCompiledTransform();
XsltSettings settingsCSV = new XsltSettings();
// 为安全原因禁用这两个属性(默认false)
settingsCSV.EnableDocumentFunction = false;
settingsCSV.EnableScript = false;
transformCSV.Load(@"..\..\PersonnelCSV.xsl", settingsCSV, resolver);

// 执行转换
XsltArgumentList xslArg = new XsltArgumentList();
CsvExtensionObject xslExt = new CsvExtensionObject();
xslArg.AddExtensionObject("urn:xslext", xslExt);
XPathDocument xPathDoc = new XPathDocument(@"..\..\Personnel.xml");
XmlWriterSettings xmlWriterSettings = new XmlWriterSettings();
xmlWriterSettings.ConformanceLevel = ConformanceLevel.Fragment;
using (XmlWriter writer = XmlWriter.Create(@"..\..\Personnel.csv",
    xmlWriterSettings))
{
    transformCSV.Transform(xPathDoc, xslArg, writer);
}
```

PersonnelCSV.xsl 样式表如下所示。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xs="http://www.w3.org/2001/XMLSchema"
                xmlns:xslext="urn:xslext">
<xsl:output method="text" encoding="UTF-8"/>
    <xsl:template match="/">
        <xsl:for-each select="Personnel">
            <xsl:for-each select="Employee">
                 <xsl:for-each select="@name">
                     <xsl:value-of
                         select="xslext:EscapeAttributeForCSV(string(.))" />
                 </xsl:for-each>,<xsl:for-each select="@title">
                     <xsl:value-of
                         select="xslext:EscapeAttributeForCSV(string(.))" />
                 </xsl:for-each>,<xsl:for-each select="@companyYears">
                     <xsl:value-of
                        select="xslext:EscapeAttributeForCSV(string(.))" />
                 </xsl:for-each>,<xsl:for-each select="@nickname">
                     <xsl:value-of
                         select="xslext:EscapeAttributeForCSV(string(.))" />
                 </xsl:for-each>
               <xsl:text> &#xd;&#xa;</xsl:text>
            </xsl:for-each>
       </xsl:for-each>
    </xsl:template>
</xsl:stylesheet>
```

从 PersonnelCSV.xls 样式表产生的输出如下所示。

```
Bob,Customer Service,1,""Happy""
Alice,Manager,12,""Business""
```

```
Chas,Salesman,3,""Money""
Rutherford,CEO,27,""BigTime""
```

我们再次做一些工作来支持 RFC 4180“常见格式和 CSV 文件 MIME 类型”。通过 CsvExtensionObject 对象上的 EscapeAttributeForCSV 方法，该对象作为 Extension 对象传递到转换中。在范例 10.8（即 10.8 节）中将会对其进行详细描述。

```
public class CsvExtensionObject
{
    public string EscapeAttributeForCSV(string attributeValue) =>
        attributeValue.Replace("\"", "\"\"");
}
```

10.6.3 讨论

XSLT 是将 XML 从一种格式转换为另一种格式的强大方法。话虽如此，LINQ 带给 C# 的执行 XML 转换而无需切换到某些解析器或进程的能力非常吸引人。这意味着，在应用程序中执行 XML 转换时，你再也不必理解 XSLT 语法或者同时维护应用程序中的 C# 和 XSLT 代码了。这还意味着当查看其他团队成员的代码时，你再也不必查看各个文件以理解转换所做的工作了；代码全部都是 C#。

这并不意味着 XSLT 作为一种转换 XML 的方法是没用或者不合时宜的；XSLT 仅仅不再是 C# 开发人员的唯一首选了。XSLT 仍能用于所有 .NET 中的现存 XML API，并且将继续使用下去。我们为你提出的挑战是，试着以 LINQ 实现现有基于 XSLT 的转换，并且亲自看到使用 LINQ 的可能性。

当使用 XSLT 执行转换时，有许多 XslCompiledTransform.Transform 方法的重写版本。因为 XmlResolver 是一个抽象类，所以需要使用 XmlUrlResolver 或 XmlSecureResolver 或传递 null 作为 XmlResolver 类型的实参。XmlUrlResolver 使用 FILE、HTTP 和 HTTPS 协议将 URL 解析为外部资源，如架构文件。XmlSecureResolver 通过要求传入凭据以限制你能够访问的资源，这将有助于防止 XML 中的跨域重定向。

> 如果你接受来自互联网的 XML，并且没有使用 XmlSecureResolver，就可以很容易地重定向到等待下载并执行的恶意 XML 代码所在的站点。如果为 XmlResolver 传递 null，那么就表明你不希望解析任何外部资源。Microsoft 已经声明 null 选项废弃，它不应当再使用，因为你总是应该使用某种类型的 XmlResolver。

XSLT 是一种功能强大的技术，允许你将 XML 转换为可以想到的任何格式，但它偶尔也可能令人心烦。XSLT 输出回车 / 换行组合的简单需求非常麻烦，我们能够找到 20 多种不同的留言板请求帮助如何完成它！在查看了针对 XSLT 的 W3C 规范后，我们发现你可以如下使用 xsl:text 元素来完成这一点。

```
<xsl:text> &#xd;&#xa;</xsl:text>
```


 代表十六进制的 13 或者一个回车，而
 代表十六进制的 10 或者一个换行。这就是来自 XML 每个雇员数据尾部的输出。

10.6.4 参考

MSDN 文档中的“XslCompiledTransform 类”“XmlResolver 类”“XmlUrlResolver 类”“XmlSecureResolver 类”和“xsl:text”主题。

10.7 验证修改过的XML文档而无需重新加载

10.7.1 问题

你正在使用 XDocument 或 XmlDocument 修改内存中加载的某个 XML 文档。一旦文档已被修改，那么需要验证修改并确保架构默认值。

10.7.2 解决方案

使用 XDocument.Validate 方法执行验证并应用模式默认值和类型信息。

使用 XML 架构文档（book.xsd）和一个 XmlReader 创建一个 XmlSchemaSet，然后使用 XDocument.Load 加载 book.xml 文件，代码如下所示。

```
// 创建架构集
XmlSchemaSet xmlSchemaSet = new XmlSchemaSet();
// 将新的架构添加到目标命名空间
//（如果存在多个架构,可以一次添加全部）
xmlSchemaSet.Add("http://tempuri.org/Book.xsd",
    XmlReader.Create(@"..\..\Book.xsd"));
XDocument book = XDocument.Load(@"..\..\Book.xml");
```

建立一个 ValidationEventHandler 以捕获所有错误，然后使用架构集合和事件处理程序调用 XDocument.Validate，根据 book.xsd 模式验证 book.xml，代码如下所示。

```
ValidationHandler validationHandler = new ValidationHandler();
ValidationEventHandler validationEventHandler =
    validationHandler.HandleValidation;
// 加载后进行验证
book.Validate(xmlSchemaSet, validationEventHandler);
```

ValidationHandler 类在 ValidateXml 属性中持有当前验证状态以及用于实现 ValidationEventHandler 的方法 HandleValidation 的代码。

```
public class ValidationHandler
{
    private object _syncRoot = new object();

    public ValidationHandler()
    {
        lock(_syncRoot)
        {
            // 设置验证初始检查为true
            this.ValidXml = true;
        }
    }
```

```
    public bool ValidXml { get; private set; }

    public void HandleValidation(object sender, ValidationEventArgs e)
    {
        lock(_syncRoot)
        {
            // 方法被调用,说明验证未通过
            ValidXml = false;
            Console.WriteLine($"Validation Error Message: {e.Message}");
            Console.WriteLine($"Validation Error Severity: {e.Severity}");
            Console.WriteLine($"Validation Error Line Number: " +
                              $"{{e.Exception?.LineNumber}");
            Console.WriteLine($"Validation Error Line Position: " +
                              $"{{e.Exception?.LinePosition}");
            Console.WriteLine($"Validation Error Source: {e.Exception?.Source}");
               Console.WriteLine($"Validation Error Source Schema: " +
                 "{e.Exception?.SourceSchemaObject}");
            Console.WriteLine($"Validation Error Source Uri: " +
                              $"{{e.Exception?.SourceUri}");
            Console.WriteLine($"Validation Error thrown from: " +
                              $"{{e.Exception?.TargetSite}");
            Console.WriteLine($"Validation Error callstack: " +
                              $"{{e.Exception?.StackTrace}");
        }
    }
}
```

如果你想知道上述代码示例中 lock 语句是什么，请查看范例 12.2（即 12.2 节）中的完整解释。简而言之，在 lock 语句中不能运行多个线程。

向 XDocument 中添加一个架构中没有的新元素节点，然后使用架构集合和事件处理程序再次调用 Validate，以重修验证修改过的 XDocument。如果文档触发了任何验证事件，那么将 ValidationHandler 实例中的 ValidationHandler.ValidXml 属性设置为 false。

```
// 添加不在架构中的新节点
// 因为我们已经验证过了,所以添加时不会触发回调
book.Root.Add(new XElement("BogusElement","Totally"));
// 现在验证新添加的内容
book.Validate(xmlSchemaSet, validationEventHandler);

if (validationHandler.ValidXml)
    Console.WriteLine("Successfully validated modified LINQ XML");
else
    Console.WriteLine("Modified LINQ XML did not validate successfully");
Console.WriteLine();
```

你还可以使用 XmlDocument.Validate 方法对 XDocument 以类似的方式执行验证，代码如下所示。

```
string xmlFile = @"..\..\Book.xml";
string xsdFile = @"..\..\Book.xsd";

// 创建架构集
```

```
XmlSchemaSet schemaSet = new XmlSchemaSet();
// 将新的架构添加到目标命名空间
// (如果存在多个架构,可以一次添加全部)
schemaSet.Add("http://tempuri.org/Book.xsd", XmlReader.Create(xsdFile));

// 加载xml文件
XmlDocument xmlDoc = new XmlDocument();
// 添加架构
xmlDoc.Schemas = schemaSet;
```

将 bool.xml 文件加载到 XmlDocument 中，建立一个 ValidationEventHandler 以捕获所有错误。然后使用事件处理程序调用 Validate，根据 book.xsd 架构验证 book.xml，代码如下所示。

```
// 加载完成之后进行验证
xmlDoc.Load(xmlFile);
ValidationHandler handler = new ValidationHandler();
ValidationEventHandler eventHandler = handler.HandleValidation;
xmlDoc.Validate(eventHandler);
```

向 XmlDocument 中添加一个架构中不存在的新元素，然后使用事件处理程序再次调用 Validate，以便重新验证修改过的 XmlDocument。如果文档触发了任何验证事件，那么将 ValidationHandler.ValidXml 属性设置为 false。

```
// 添加不在架构中的新节点
// 因为我们已经验证过了,所以添加时不会触发回调
XmlNode newNode = xmlDoc.CreateElement("BogusElement");
newNode.InnerText = "Totally";
// 添加新元素
xmlDoc.DocumentElement.AppendChild(newNode);
// 现在验证新添加的内容
xmlDoc.Validate(eventHandler);

if (handler.ValidXml)
    Console.WriteLine("Successfully validated modified XML");
else
    Console.WriteLine("Modified XML did not validate successfully");
```

10.7.3 讨论

使用 XmlDocument 而不是 XDocument 的一个好处在于，存在一个 XmlDocument.Validate 的重载，允许传入一个特定的 XmlNode 进行验证。XDocument 上不存在如此细粒度的控制。

```
public void Validate(
    ValidationEventHandler validationEventHandler,
    XmlNode nodeToValidate
);
```

解决这一问题的另一个方法是使用 XmlDocument 实例化 XmlNodeReader 的一个实例，然后使用验证设置创建一个 XmlReader，如范例 10.3（即 10.3 节）所示。当读取器导航下列 XML 时，它允许进行连续验证。

运行该代码的输出如下所示。

```
Validation Error Message: The element 'Book' in namespace 'http://tempuri.org/Bo
ok.xsd' has invalid child element 'BogusElement'. List of possible elements expe
cted: 'Chapter' in namespace 'http://tempuri.org/Book.xsd'.
Validation Error Severity: Error
Validation Error Line Number: 0
Validation Error Line Position: 0
Validation Error Source:
Validation Error Source Schema:
Validation Error Source Uri:
Validation Error thrown from:
Validation Error callstack:
Modified LINQ XML did not validate successfully

Validation Error Message: The element 'Book' in namespace 'http://tempuri.org/Bo
ok.xsd' has invalid child element 'BogusElement'. List of possible elements expe
cted: 'Chapter' in namespace 'http://tempuri.org/Book.xsd'
Validation Error Severity: Error
Validation Error Line Number: 0
Validation Error Line Position: 0
Validation Error Source:
Validation Error Source Schema:
Validation Error Source Uri: file:///C:/CSCB6/CSharpRecipes/Book.xml
Validation Error thrown from:
Validation Error callstack:
Modified XML did not validate successfully
```

要注意，用户添加的 `BogusElement` 元素不是用于 Book 元素的架构组成部分，因此你得到了一个带有关于错误发生地点信息的验证错误事件。最后，将针对得到的一个报告，说明修改的 XML 未能正确地验证。

10.7.4 参考

范例 10.2（即 10.2 节）；MSDN 文档中的“`XDocument` 类”和“`XmlDocument.Validate`”主题。

10.8 扩展转换

10.8.1 问题

你希望执行超出转换技术范围之外的操作，以在转换后的结果中包含数据。

10.8.2 解决方案

如果你使用 LINQ to XML，那么可以在转换结果集时直接调用一个函数，下面是对 `GetErrata` 的调用。

```
XElement publications = XElement.Load(@"..\..\publications.xml");
XElement transformedPublications =
```

```
        new XElement("PublishedWorks",
            from b in publications.Elements("Book")
            select new XElement(b.Name,
                        new XAttribute(b.Attribute("name")),
                        from c in b.Elements("Chapter")
                        select new XElement("Chapter", GetErrata(c))));
    Console.WriteLine(transformedPublications.ToString());
    Console.WriteLine();
```

上述示例中使用的 `GetErrata` 方法如下所示。

```
    private static XElement GetErrata(XElement chapter)
    {
        // 此处我们可以进行其他的查找操作(XML、数据库和Web服务)
        // 以获得信息并添加到转换结果中
        string errata = $"{chapter.Value} has {chapter.Value.Length} errata";
        return new XElement("Errata", errata);
    }
```

如果你使用 XSLT，那么可以向转换添加一个扩展对象，它可以根据传入的节点执行所需的操作。通过使用 `XsltArgumentList.AddExtensionObject` 方法可以达成这一工作。你创建的对象（`XslExtensionObject`）可在 XSLT 中进行访问，调用其上的方法可返回你希望在最终转换结果中包含的数据。

```
    string xmlFile = @"..\..\publications.xml";
    string xslt = @"..\..\publications.xsl";

    // 创建XslCompiledTransform并加载样式表
    XslCompiledTransform transform = new XslCompiledTransform();
    transform.Load(xslt);
    // 加载xml
    XPathDocument xPathDoc = new XPathDocument(xmlFile);

    // 使用扩展对象创建样式表参数
    XsltArgumentList xslArg = new XsltArgumentList();
    XslExtensionObject xslExt = new XslExtensionObject();
    xslArg.AddExtensionObject("urn:xslext", xslExt);

    // 将输出发送到控制台并执行转换
    using (XmlWriter writer = XmlWriter.Create(Console.Out))
    {
        transform.Transform(xPathDoc, xslArg, writer);
    }
```

要注意，当扩展对象添加到 `XsltArgumentList` 中时，它提供了一个 `urn:xslext` 命名空间。该命名空间在 XSLT 样式表中使用，以引用到该对象。`XSLExtensionObject` 的定义如下所示。

```
    // 为功能提供帮助的扩展对象
    public class XslExtensionObject
    {
        public XPathNodeIterator GetErrata(XPathNodeIterator nodeChapter)
        {
            // 此处我们可以进行其他的查找操作(XML、数据库和Web服务)
```

```
            // 以获得信息并添加到转换结果中
            nodeChapter.MoveNext();
            string errata = $"<Errata>{nodeChapter.Current.Value} has " +
                $"{nodeChapter.Current.Value.Length} errata</Errata>";
            XmlDocument xDoc = new XmlDocument();
            xDoc.LoadXml(errata);
            XPathNavigator xPathNav = xDoc.CreateNavigator();
            xPathNav.MoveToChild(XPathNodeType.Element);
            XPathNodeIterator iter = xPathNav.Select(".");
            return iter;
        }
    }
```

在执行 XSLT 样式表期间，调用 GetErrata 方法向转换提供 XPathNodeIterator 格式的数据。xmlns:xslext 命名空间被声明为 urn:xslext，它匹配作为转换的实参传入的命名空间值。在处理用于每个 Chapter 的 Book 模板时，使用包含对 xslext:GetErrata 方法的调用的 select 标准来调用 xsl:value-of。该样式表如下所示。

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xslext="urn:xslext">
    <xsl:template match="/">
        <xsl:element name="PublishedWorks">
            <xsl:apply-templates/>
        </xsl:element>
    </xsl:template>
    <xsl:template match="Book">
        <Book>
            <xsl:attribute name ="name">
                <xsl:value-of select="@name"/>
            </xsl:attribute>
            <xsl:for-each select="Chapter">
                <Chapter>
                    <xsl:value-of select="xslext:GetErrata(/)"/>
                </Chapter>
            </xsl:for-each>
        </Book>
    </xsl:template>
</xsl:stylesheet>
```

这两种方法输出相同，看起来如下所示（部分清单）。

```
<PublishedWorks>
  <Book name="Subclassing and Hooking with Visual Basic">
    <Chapter>
      <Errata>Introduction has 12 errata</Errata>
    </Chapter>
    ...
  </Book>
  <Book name="C# Cookbook">
    <Chapter>
      <Errata>Numbers has 7 errata</Errata>
    </Chapter>
    ...
  </Book>
```

```
  <Book name="C# Cookbook 2.0">
    <Chapter>
      <Errata>Numbers and Enumerations has 24 errata</Errata>
    </Chapter>
    ...
  </Book>
  <Book name="C# 3.0 Cookbook">
    <Chapter>
      <Errata>Language Integrated Query (LINQ) has 32 errata</Errata>
    </Chapter>
    ...
  </Book>
  <Book name="C# 6.0 Cookbook">
    <Chapter>
      <Errata>Classes and Generics has 20 errata</Errata>
    </Chapter>
    <Chapter>
      <Errata>Collections, Enumerators, and Iterators has 39 errata</Errata>
    </Chapter>
    <Chapter>
      <Errata>Data Types has 10 errata</Errata>
    </Chapter>
    <Chapter>
      <Errata>LINQ and Lambda Expressions has 27 errata</Errata>
    </Chapter>
    <Chapter>
      <Errata>Debugging and Exception Handling has 32 errata</Errata>
    </Chapter>
    <Chapter>
      <Errata>Reflection and Dynamic Programming has 34 errata</Errata>
    </Chapter>
    <Chapter>
      <Errata>Regular Expressions has 19 errata</Errata>
    </Chapter>
    <Chapter>
      <Errata>Filesystem I/O has 14 errata</Errata>
    </Chapter>
    <Chapter>
      <Errata>Networking and Web has 18 errata</Errata>
    </Chapter>
    <Chapter>
      <Errata>XML has 3 errata</Errata>
    </Chapter>
    <Chapter>
      <Errata>Security has 8 errata</Errata>
    </Chapter>
    <Chapter>
      <Errata>Threading, Synchronization, and Concurrency has 43 errata</Errata>
    </Chapter>
    <Chapter>
      <Errata>Toolbox has 7 errata</Errata>
    </Chapter>
  </Book>
</PublishedWorks>
```

10.8.3 讨论

使用 LINQ to XML，你可以扩展自己的转换代码，通过简单地添加知道如何操作和返回 XElement 的方法调用，以包含额外的逻辑。它简单地将另一个方法调用添加到生成结果集的查询中，而且不会因调用带来任何额外的性能问题。当然，如果操作的开销大，那么也可能会令转换变慢，但这个问题在测试你的代码时很容易定位。

调用来自一个 XSLT 样式表内部的用户定制代码的能力是非常强大的，但使用时应当小心。如下向样式表中添加代码通常会令它们在其他环境中变得无用。如果不必使用样式表在其他解析器中转换 XML，那么对于以常规 XSLT 语法难以或无法完成的工作，它可能是一种好方法。

解决方案使用的示例数据如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<Publications>
  <Book name="Subclassing and Hooking with Visual Basic">
    <Chapter>Introduction</Chapter>
    <Chapter>Windows System-Specific Information</Chapter>
    <Chapter>The Basics of Subclassing and Hooks</Chapter>
    <Chapter>Subclassing and Superclassing</Chapter>
    <Chapter>Subclassing the Windows Common Dialog Boxes</Chapter>
    <Chapter>ActiveX Controls and Subclassing</Chapter>
    <Chapter>Superclassing</Chapter>
    <Chapter>Debugging Techniques for Subclassing</Chapter>
    <Chapter>WH_CALLWNDPROC</Chapter>
    <Chapter>WH_CALLWNDPROCRET</Chapter>
    <Chapter>WH_GETMESSAGE</Chapter>
    <Chapter>WH_KEYBOARD and WH_KEYBOARD_LL</Chapter>
    <Chapter>WH_MOUSE and WH_MOUSE_LL</Chapter>
    <Chapter>WH_FOREGROUNDIDLE</Chapter>
    <Chapter>WH_MSGFILTER</Chapter>
    <Chapter>WH_SYSMSGFILTER</Chapter>
    <Chapter>WH_SHELL</Chapter>
    <Chapter>WH_CBT</Chapter>
    <Chapter>WH_JOURNALRECORD</Chapter>
    <Chapter>WH_JOURNALPLAYBACK</Chapter>
    <Chapter>WH_DEBUG</Chapter>
    <Chapter>Subclassing .NET WinForms</Chapter>
    <Chapter>Implementing Hooks in VB.NET</Chapter>
  </Book>
  <Book name="C# Cookbook">
    <Chapter>Numbers</Chapter>
    <Chapter>Strings and Characters</Chapter>
    <Chapter>Classes And Structures</Chapter>
    <Chapter>Enums</Chapter>
    <Chapter>Exception Handling</Chapter>
    <Chapter>Diagnostics</Chapter>
    <Chapter>Delegates and Events</Chapter>
    <Chapter>Regular Expressions</Chapter>
    <Chapter>Collections</Chapter>
    <Chapter>Data Structures and Algorithms</Chapter>
    <Chapter>File System IO</Chapter>
```

```
  <Chapter>Reflection</Chapter>
  <Chapter>Networking</Chapter>
  <Chapter>Security</Chapter>
  <Chapter>Threading</Chapter>
  <Chapter>Unsafe Code</Chapter>
  <Chapter>XML</Chapter>
</Book>
<Book name="C# Cookbook 2.0">
  <Chapter>Numbers and Enumerations</Chapter>
  <Chapter>Strings and Characters</Chapter>
  <Chapter>Classes And Structures</Chapter>
  <Chapter>Generics</Chapter>
  <Chapter>Collections</Chapter>
  <Chapter>Iterators and Partial Types</Chapter>
  <Chapter>Exception Handling</Chapter>
  <Chapter>Diagnostics</Chapter>
  <Chapter>Delegates, Events, and Anonymous Methods</Chapter>
  <Chapter>Regular Expressions</Chapter>
  <Chapter>Data Structures and Algorithms</Chapter>
  <Chapter>File System IO</Chapter>
  <Chapter>Reflection</Chapter>
  <Chapter>Web</Chapter>
  <Chapter>XML</Chapter>
  <Chapter>Networking</Chapter>
  <Chapter>Security</Chapter>
  <Chapter>Threading and Synchronization</Chapter>
  <Chapter>Unsafe Code</Chapter>
  <Chapter>Toolbox</Chapter>
</Book>
<Book name="C# 3.0 Cookbook">
  <Chapter>Language Integrated Query (LINQ)</Chapter>
  <Chapter>Strings and Characters</Chapter>
  <Chapter>Classes And Structures</Chapter>
  <Chapter>Generics</Chapter>
  <Chapter>Collections</Chapter>
  <Chapter>Iterators, Partial Types, and Partial Methods </Chapter>
  <Chapter>Exception Handling</Chapter>
  <Chapter>Diagnostics</Chapter>
  <Chapter>Delegates, Events, and Lambda Expressions</Chapter>
  <Chapter>Regular Expressions</Chapter>
  <Chapter>Data Structures and Algorithms</Chapter>
  <Chapter>File System IO</Chapter>
  <Chapter>Reflection</Chapter>
  <Chapter>Web</Chapter>
  <Chapter>XML</Chapter>
  <Chapter>Networking</Chapter>
  <Chapter>Security</Chapter>
  <Chapter>Threading and Synchronization</Chapter>
  <Chapter>Toolbox</Chapter>
  <Chapter>Numbers and Enumerations</Chapter>
</Book>
<Book name="C# 6.0 Cookbook">
  <Chapter>Classes and Generics</Chapter>
  <Chapter>Collections, Enumerators, and Iterators</Chapter>
  <Chapter>Data Types</Chapter>
```

```
    <Chapter>LINQ and Lambda Expressions</Chapter>
    <Chapter>Debugging and Exception Handling</Chapter>
    <Chapter>Reflection and Dynamic Programming</Chapter>
    <Chapter>Regular Expressions</Chapter>
    <Chapter>Filesystem I/O</Chapter>
    <Chapter>Networking and Web</Chapter>
    <Chapter>XML</Chapter>
    <Chapter>Security</Chapter>
    <Chapter>Threading, Synchronization, and Concurrency</Chapter>
    <Chapter>Toolbox</Chapter>
  </Book>
</Publications>
```

10.8.4　参考

MSDN 文档中的“使用 LINQ 进行数据转换”和“XsltArgumentList 类”主题。

10.9　从现有XML文件批量获取架构

10.9.1　问题

你进入了一个新项目，在其中使用 XML 进行数据传输，但在你之前到来的程序员由于某种原因未使用 XSD。你需要为每个 XML 示例生成开始的架构文件。

10.9.2　解决方案

使用 XmlSchemaInference 类从 XML 示例中推导出架构。例 10-7 中的 GenerateSchemasForDirectory 函数枚举了给定目录中的所有 XML 文件并使用 GenerateSchemasForFile 方法处理每一个文件。GenerateSchemasForFile 使用 XmlSchemaInference.InferSchema 方法获得针对给定 XML 文件的架构。一旦确定了所有架构，那么 GenerateSchemasForFile 会遍历集合，并使用 FileStream 将每个架构保存成一个 XSD 文件。

例 10-7：生成一个 XML 架构

```
public static void GenerateSchemasForFile(string file)
{
    // 设置此文件的读取器
    using (XmlReader reader = XmlReader.Create(file))
    {
        XmlSchemaSet schemaSet = new XmlSchemaSet();
        XmlSchemaInference schemaInference =
                        new XmlSchemaInference();

        // 获得架构
        schemaSet = schemaInference.InferSchema(reader);

        string schemaPath = string.Empty;
        foreach (XmlSchema schema in schemaSet.Schemas())
        {
            // 创建架构文件并将架构写到文件中
```

```
                schemaPath = $"{Path.GetDirectoryName(file)}\\" +
                                $"{Path.GetFileNameWithoutExtension(file)}.xsd";
                using (FileStream fs =
                    new FileStream(schemaPath, FileMode.OpenOrCreate))
                {
                    schema.Write(fs);
                    fs.Flush();
                }
            }
        }
    }

    public static void GenerateSchemasForDirectory(string dir)
    {
        // 确认目录存在
        if (Directory.Exists(dir))
        {
            // 获得目录中的文件列表
            string[] files = Directory.GetFiles(dir, "*.xml");
            foreach (string file in files)
            {
                GenerateSchemasForFile(file);
            }
        }
    }
```

可以如下调用 GenerateSchemasForDirectory 方法。

```
// 获得当前运行目录的向上两个级别的目录
DirectoryInfo di = new DirectoryInfo(@"..\..");
string dir = di.FullName;
// 生成架构
GenerateSchemasForDirectory(dir);
```

10.9.3　讨论

拥有用于某一应用程序中的 XML 文件的 XSD，能够进行以下几项工作。

- 验证系统中存在的 XML
- 用文档记录数据语义
- 通过 XML 读取方法以编程方式发现数据结构

使用 GenerateSchemasForFile 方法可以快速启动开发 XML 架构的过程，但每个架构应当由负责产生 XML 的团队成员进行复查。这有助于确保架构声明的规则是正确的，还可以确保添加诸如模式默认值的附加项和其他关系。示例 XML 文档中未存在的所有关系将被架构生成器忽略。

10.9.4　参考

MSDN 文档中的“XmlSchemaInference 类”和“XML 架构（XSD）参考”主题。

10.10 将参数传递给转换

10.10.1 问题

你需要使用最常用的模式转换某些数据。对于少数在转换过程中可能发生改变的数据项，你不希望对每种变化都使用一个单独的机制。

10.10.2 解决方案

如果用户使用 LINQ to XML，那么可以简单地构建一个方法，封装转换代码并向方法传递参数，就像你对其他代码通常所做的一样。

```
// 使用LINQ而非XSLT转换数据
string storeTitle = "Hero Comics Inventory";
string pageDate = DateTime.Now.ToString("F");
XElement parameterExample = XElement.Load(@"..\..\ParameterExample.xml");
string htmlPath = @"..\..\ParameterExample_LINQ.htm";
TransformWithParameters(storeTitle, pageDate, parameterExample, htmlPath);

// 现在改变参数
storeTitle = "Fabulous Adventures Inventory";
pageDate = DateTime.Now.ToString("D");
htmlPath = @"..\..\ParameterExample2_LINQ.htm";
TransformWithParameters(storeTitle, pageDate, parameterExample, htmlPath);
```

`TransformWithParameters` 方法如下所示。

```
private static void TransformWithParameters(string storeTitle, string pageDate,
    XElement parameterExample, string htmlPath)
{
    XElement transformedParameterExample =
        new XElement("html",
            new XElement("head"),
            new XElement("body",
                new XElement("h3", $"Brought to you by {storeTitle} " +
                        $"on {pageDate}{Environment.NewLine}"),
                new XElement("br"),
                new XElement("table",
                    new XAttribute("border","2"),
                    new XElement("thead",
                        new XElement("tr",
                            new XElement("td",
                                new XElement("b","Heroes")),
                            new XElement("td",
                                new XElement("b","Edition")))),
                    new XElement("tbody",
                        from cb in parameterExample.Elements("ComicBook")
                        orderby cb.Attribute("name").Value descending
                        select new XElement("tr",
                                new XElement("td",cb.Attribute("name").Value),
                                new XElement("td",
                                    cb.Attribute("edition").Value))))));
```

```
    transformedParameterExample.Save(htmlPath);
}
```

如果你使用 XSLT 执行转换，那么可使用 XsltArgumentList 类向 XSLT 转换传递参数。这一技术允许程序生成用于样式表的对象（例如一个动态字符串），当样式表转换给定 XML 文档时访问此对象。在下列示例中，storeTitle 和 pageDate 参数被传递给转换。storeTitle 用于连环画商店的店名，pageDate 是运行报告的日期。使用 XsltArgumentList 对象的实例 args 的 AddParam 方法添加它们。

```
// 使用XSLT和参数转换
XsltArgumentList args = new XsltArgumentList();
args.AddParam("storeTitle", "", "Hero Comics Inventory");
args.AddParam("pageDate", "", DateTime.Now.ToString("F"));

// 使用默认凭据创建一个解析器
XmlUrlResolver resolver = new XmlUrlResolver();
resolver.Credentials = System.Net.CredentialCache.DefaultCredentials;
```

XsltSettings 类允许修改转换的行为。如果你使用 XsltSettings.Default 实例，那么在转换时不允许编写脚本或者使用 document XSLT 函数，因为它们可能存在安全风险。如果样式表来自一个可信源，你可以创建一个 XsltSettings 对象并使用，但是还是安全一些好。对代码进一步修改可以令其使用不可信的 XSLT 样式表。

```
XslCompiledTransform transform = new XslCompiledTransform();
// 加载样式表
transform.Load(@"..\..\ParameterExample.xslt", XsltSettings.Default,
    resolver);
// 执行转换
FileStream fs = null;
using (fs =
    new FileStream(@"..\..\ParameterExample.htm",
                   FileMode.OpenOrCreate, FileAccess.Write))
{
    transform.Transform(@"..\..\ParameterExample.xml", args, fs);
}     XslCompiledTransform transform = new XslCompiledTransform();
    // Load up the stylesheet.
    transform.Load(@"..\..\ParameterExample.xslt", XsltSettings.Default,
        resolver);

    // Perform the transformation.
    FileStream fs = null;
    using (fs = new FileStream(@"..\..\ParameterExample.htm",
         FileMode.OpenOrCreate, FileAccess.Write))
    {
        transform.Transform(@"..\..\ParameterExample.xml", args, fs);
    }
```

为了显示使用不同参数的情况，现在你可以修改 storeTitle 和 pageDate 并再次运行转换。

```
// 现在修改参数并重新处理
args = new XsltArgumentList();
args.AddParam("storeTitle", "", "Fabulous Adventures Inventory");
args.AddParam("pageDate", "", DateTime.Now.ToString("D"));
```

```
using (fs = new FileStream(@"..\..\ParameterExample2.htm",
    FileMode.OpenOrCreate, FileAccess.Write))
{
    transform.Transform(@"..\..\ParameterExample.xml", args, fs);
}
```

ParameterExample.xml 文件包含下列内容。

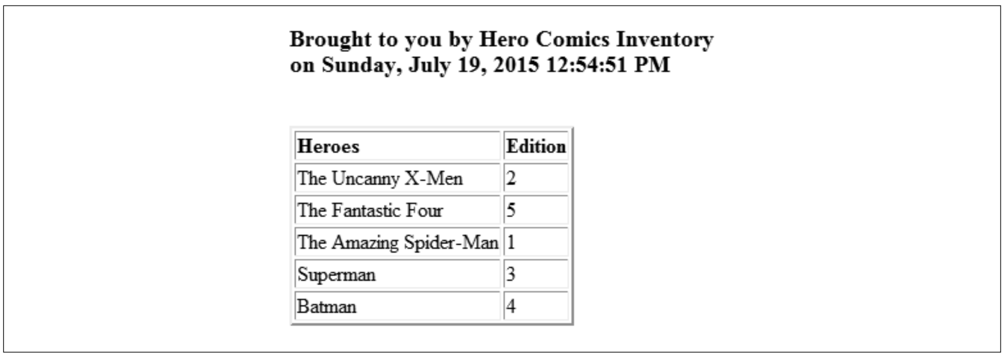
```
<?xml version="1.0" encoding="utf-8" ?>
<?xml-stylesheet href="ParameterExample.xslt" type="text/xsl"?>
<ParameterExample>
    <ComicBook name="The Amazing Spider-Man" edition="1"/>
    <ComicBook name="The Uncanny X-Men" edition="2"/>
    <ComicBook name="Superman" edition="3"/>
    <ComicBook name="Batman" edition="4"/>
    <ComicBook name="The Fantastic Four" edition="5"/>
</ParameterExample>
```

ParameterExample.xlst 文件包含下列内容。

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html" indent="yes" />
  <xsl:param name="storeTitle"/>
  <xsl:param name="pageDate"/>
  <xsl:template match="ParameterExample">
    <html>
      <head/>
      <body>
        <h3>
          <xsl:text>Brought to you by </xsl:text>
          <xsl:value-of select="$storeTitle"/>
          <xsl:text> on </xsl:text>
          <xsl:value-of select="$pageDate"/>
          <xsl:text> &#xd;&#xa;</xsl:text>
        </h3>
        <br/>
        <table border="2">
          <thead>
            <tr>
              <td>
                <b>Heroes</b>
              </td>
              <td>
                <b>Edition</b>
              </td>
            </tr>
          </thead>
          <tbody>
            <xsl:apply-templates/>
          </tbody>
        </table>
      </body>
    </html>
  </xsl:template>
  <xsl:template match="ComicBook">
```

```
        <tr>
          <td>
            <xsl:value-of select="@name"/>
          </td>
          <td>
            <xsl:value-of select="@edition"/>
          </td>
        </tr>
      </xsl:template>
    </xsl:stylesheet>
```

使用 XSLT 到 ParameterExample.htm 或者使用 LINQ 到 ParameterExample_LINQ.htm 的第一次转换的输出如图 10-2 所示。

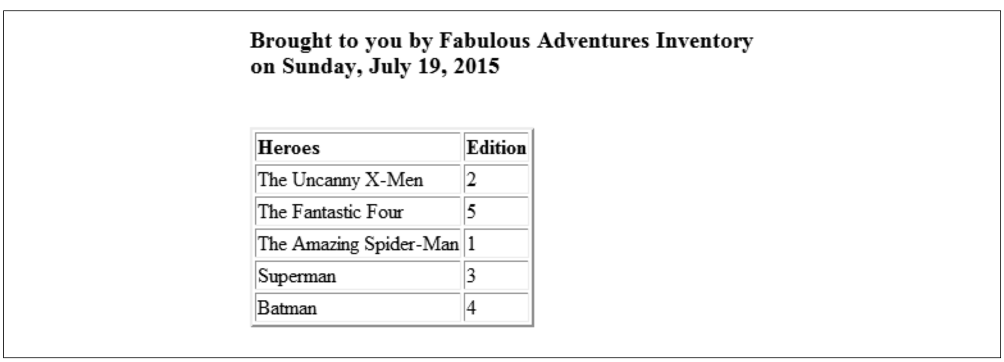


Brought to you by Hero Comics Inventory
on Sunday, July 19, 2015 12:54:51 PM

Heroes	Edition
The Uncanny X-Men	2
The Fantastic Four	5
The Amazing Spider-Man	1
Superman	3
Batman	4

图 10-2：第一组参数的输出

使用 XSLT 到 ParameterExample2.htm 或者使用 LINQ 到 ParameterExample2_LINQ.htm 的第二次转换的输出如图 10-3 所示。



Brought to you by Fabulous Adventures Inventory
on Sunday, July 19, 2015

Heroes	Edition
The Uncanny X-Men	2
The Fantastic Four	5
The Amazing Spider-Man	1
Superman	3
Batman	4

图 10-3：第二组参数的输出

10.10.3 讨论

两种方法都允许你模板化代码并提供参数以修改输出。使用 LINQ to XML 方法，代码都是 .NET 的，而 .NET 分析工具可用于度量转换的影响。使用代码的声明式风格传达的意图

比必须转到外部 XSLT 文件更明显。如果你不了解 XSLT，那么就不必学习它，因为你现在就可以用代码完成这项工作。

如果你已经了解 XSLT，那么可以继续充分利用它。向 XSLT 样式表传递信息的能力在通过 XSLT 转换设计报告或用户界面时允许更大程度的灵活性。这种能力有助于根据你能想到的任何标准来定制输出，因为传入的数据完全由程序来控制。一旦你掌握了使用带参数的 XSLT 的方法，那么一种全新的自定义级别就成为了可能。一个额外的好处是，它在不同环境之间是可移植的。

10.10.4 参考

MSDN 文档中的“使用 LINQ 进行数据转换”“`XsltArgumentList` 类”和“`XsltSettings` 类”主题。

第11章
安全

11.0　简介

在 .NET 中运行代码的安全性围绕*代码访问安全性*（code access security，CAS）展开。CAS 基于程序集的来源和程序集自身的特性决定其可信赖性，比如程序集的散列值。例如，计算机本身安装的代码要比从网上下载的代码更加安全。在允许代码运行之前，运行时同样会验证程序集的元数据及其类型安全性。

在 .NET Framework 中有很多种编写安全代码和保护数据的机制。在本章中，我们将探讨以下主题：控制对类型的访问，加密和解密，用于安全的随机数，安全存储数据，以及使用编程方式和声明方式安全等。

11.1　加密和解密字符串

11.1.1　问题

你有一个希望加密或解密的字符串（或许是密码或软件密钥），该字符串会以某种形式存储，例如存储在文件或注册表中。你希望能将这些字符串保密，从而使其他用户无法得到这些信息。

11.1.2　解决方案

给字符串加密将有助于防止用户读取和破译这些信息。例 11-1 中所示的 `CryptoString` 类包含两个静态方法，用来加密和解密一个字符串；以及两个静态属性，在加密之后获得生成的密钥和初始向量（initialization vector，IV；是用作加密数据起点的一个随机数）。

例 11-1：CryptoString 类

```
using System;
using System.Security.Cryptography;

public sealed class CryptoString
{
    private CryptoString() {}

    private static byte[] savedKey = null;
    private static byte[] savedIV = null;

    public static byte[] Key { get; set; }

    public static byte[] IV { get; set; }

    private static void RdGenerateSecretKey(RijndaelManaged rdProvider)
    {
        if (savedKey == null)
        {
            rdProvider.KeySize = 256;
            rdProvider.GenerateKey();
            savedKey = rdProvider.Key;
        }
    }

    private static void RdGenerateSecretInitVector(RijndaelManaged rdProvider)
    {
        if (savedIV == null)
        {
            rdProvider.GenerateIV();
            savedIV = rdProvider.IV;
        }
    }

    public static string Encrypt(string originalStr)
    {
        // 对将要存储在内存中的数据字符串进行加密
        byte[] originalStrAsBytes = Encoding.ASCII.GetBytes(originalStr);
        byte[] originalBytes = {};

        // 创建MemoryStream以包含输出
        using (MemoryStream memStream = new
                 MemoryStream(originalStrAsBytes.Length))
        {
            using (RijndaelManaged rijndael = new RijndaelManaged())
            {
                // 生成并保存密钥和初始向量
                RdGenerateSecretKey(rijndael);
                RdGenerateSecretInitVector(rijndael);

                if (savedKey == null || savedIV == null)
                {
                    throw (new NullReferenceException(
                            "savedKey and savedIV must be non-null."));
                }
```

```
            // 创建加密器和流对象
            using (ICryptoTransform rdTransform =
                   rijndael.CreateEncryptor((byte[])savedKey.
                   Clone(),(byte[])savedIV.Clone()))
            {
                using (CryptoStream cryptoStream = new CryptoStream(memStream,
                     rdTransform, CryptoStreamMode.Write))
                {
                    // 将加密后的数据写入MemoryStream
                    cryptoStream.Write(originalStrAsBytes, 0,
                               originalStrAsBytes.Length);
                    cryptoStream.FlushFinalBlock();
                    originalBytes = memStream.ToArray();
                }
            }
        }
    }
    // 转换加密过的字符串
    string encryptedStr = Convert.ToBase64String(originalBytes);
    return (encryptedStr);
}

public static string Decrypt(string encryptedStr)
{
    // 逆转换加密字符串
    byte[] encryptedStrAsBytes = Convert.FromBase64String(encryptedStr);
    byte[] initialText = new Byte[encryptedStrAsBytes.Length];

    using (RijndaelManaged rijndael = new RijndaelManaged())
    {
        using (MemoryStream memStream = new MemoryStream(encryptedStrAsBytes))
        {
            if (savedKey == null || savedIV == null)
            {
                throw (new NullReferenceException(
                        "savedKey and savedIV must be non-null."));
            }

            // 创建解密器和流对象
            using (ICryptoTransform rdTransform =
                 rijndael.CreateDecryptor((byte[])savedKey.Clone(),
                 (byte[])savedIV.Clone()))
            {
               using (CryptoStream cryptoStream = new CryptoStream(memStream,
                rdTransform, CryptoStreamMode.Read))
                {
                    // 将解密过的字符串作为byte[]读取
                    cryptoStream.Read(initialText, 0, initialText.Length);
                }
            }
        }
    }

    // 将byte[]转换为string
```

```
            string decryptedStr = Encoding.ASCII.GetString(initialText);
            return (decryptedStr);
        }
    }
```

11.1.3 讨论

CryptoString 类除了私有实例构造函数外只包含静态成员，用来阻止任何人直接从该类创建对象。

该类使用 Rijndael 算法（Rijndael algorithm）对字符串进行加密和解密。该算法可在 System.Security.Cryptography.RijndaelManaged 类中找到。该算法要求一个密钥和一个初始向量，两者都是 byte 数组。通过调用 RijndaelManaged 类上的 GenerateKey 方法可以生成一个随机密钥。该方法不接受参数，返回 void。生成的密钥被放在 RijndaelManaged 类的 Key 属性中。GenerateIV 方法生成一个随机初始向量，放在 RijndaelManaged 类的 IV 属性中。

Key 和 IV 属性中的 byte 数组必须存储起来以便随后使用，并且不能进行更改。这是由私有密钥类的特性决定的，例如 RijndaelManaged。Key 和 IV 值必须由加密和解密程序使用，以成功地加密和解密数据。

SavedKey 和 SavedIV 私有静态字段分别包含密钥和初始向量。加密和解密方法都使用密钥来加密和解密数据。这就是这两个值作为公共属性的原因，因此它们可以存储在其他安全地方以备之后使用。这意味着任何由该对象加密的字符串必须由该对象进行解密。初始向量使得从加密字符串中推演出密钥更加困难。为了实现这一点，初始向量使两个同样的加密字符串（使用相同的密钥加密）的加密形式看起来有巨大差异。

CryptoString 类中的两个方法 RdGenerateSecretKey 和 RdGenerateSecretInitVector 用于在密钥和初始向量皆不存在时生成它们。RdGenerateSecretKey 方法生成密钥，存放在 SavedKey 字段中。同样，RdGenerateSecretInitVector 生成初始向量，存放在 SavedIV 字段中。仅为该类生成了唯一的密钥和 IV。这使得加密和解密程序能够一直访问相同的密钥和 IV 信息。

CryptoString 类的 Encrypt 和 Decrypt 方法执行加密和解密一个字符串的实际工作。Encrypt 方法接受一个希望加密的字符串并返回一个加密的字符串。下列代码调用该方法并传递一个要加密的字符串。

```
string encryptedString = CryptoString.Encrypt("MyPassword");
Console.WriteLine($"encryptedString: {encryptedString}");

// 获得所使用的密钥和IV,以便稍后可以解密
byte [] key = CryptoString.Key;
byte [] IV = CryptoString.IV;
```

一旦字符串被加密，就存储密钥和 IV 用于以后进行解密。该方法显示的内容如下所示。

```
encryptedString: NmmKqBO4iPT+BDxgLVwzgQ==
```

要注意，你的输出可能会有所不同，因为你将会使用一个不同的密钥和 IV 值。下列代码

设置用于加密字符串的密钥和 IV，然后调用 Decrypt 方法解密前面加密过的字符串。

```
CryptoString.Key = key;
CryptoString.IV = IV;
string decryptedString = CryptoString.Decrypt(encryptedString);
Console.WriteLine($"decryptedString: {decryptedString}");
```

该方法显示的内容如下所示。

```
decryptedString: MyPassword
```

在要加密的字符串中使用诸如 \r、\n、\r\n 或 \t 等转义序列看起来不会出现什么问题。此外，不管带不带转义字符，使用加引号的字符串文本都不会出现问题，如下所示。

```
@"MyPassword"
```

11.1.4　参考

范例 11.2（即 11.2 节）；MSDN 文档中的“System.Cryptography 命名空间”“MemoryStream 类”“ICryptoTransform 接口”和“RijndaelManaged 类”主题。

11.2　加密和解密文件

11.2.1　问题

你有一些敏感信息，必须在将其写入一个可能位于不安全区域的文件之前进行加密。该信息在被读取回应用程序之前也必须进行解密。

11.2.2　解决方案

使用多种加密提供者将数据以加密格式写入文件。下面的类可以完成这项工作，它有一个构造函数，接受 System.Security.Cryptography.SymmetricAlgorithm 类的一个实例和文件路径。SymmetricAlgorithm 类是 .NET 中用于加密提供者的抽象基类，因此你能够确保该类可以被扩展并覆盖到全部的密码算法。该示例实现了对 TripleDES 和 Rijndael 的支持。

该解决方案需要使用以下命名空间。

```
using System;
using System.Text;
using System.IO;
using System.Security.Cryptography;
```

SecretFile 类（参考例 11-2）可用于 TripleDES，如下所示。

```
// 使用TripleDES
using (TripleDESCryptoServiceProvider tdes = new
       TripleDESCryptoServiceProvider())
{
    SecretFile secretTDESFile = new SecretFile(tdes,"tdestext.secret");
```

```
        string encrypt = "My TDES Secret Data!";
        Console.WriteLine($"Writing secret data: {encrypt}");
        secretTDESFile.SaveSensitiveData(encrypt);

        // 保存到存储中以读取文件
        byte [] key = secretTDESFile.Key;
        byte [] IV = secretTDESFile.IV;

        string decrypt = secretTDESFile.ReadSensitiveData();
        Console.WriteLine($"Read secret data: {decrypt}");
    }
```

要将 Rijndael 用于 SecretFile，只需像下面这样替代构造函数中的提供者即可。

```
    // 使用Rijndael
    using (RijndaelManaged rdProvider = new RijndaelManaged())
    {
        SecretFile secretRDFile = new SecretFile(rdProvider,"rdtext.secret");

        string encrypt = "My Rijndael Secret Data!";

        Console.WriteLine($"Writing secret data: {encrypt}");
        secretRDFile.SaveSensitiveData(encrypt);
        // 保存到存储中以读取文件
        byte [] key = secretRDFile.Key;
        byte [] IV = secretRDFile.IV;

        string decrypt = secretRDFile.ReadSensitiveData();
        Console.WriteLine($"Read secret data: {decrypt}");
    }
```

例 11-2 展示了 SecretFile 的实现。

例 11-2：SecretFile 类

```
    public class SecretFile
    {
        private byte[] savedKey = null;
        private byte[] savedIV = null;
        private SymmetricAlgorithm symmetricAlgorithm;
        string path;

        public byte[] Key { get; set; }

        public byte[] IV { get; set; }

        public SecretFile(SymmetricAlgorithm algorithm, string fileName)
        {
            symmetricalgorithm;
            path = fileName;
        }

        public void SaveSensitiveData(string sensitiveData)
        {
            // 将数据字符串编码以存储到加密文件中
            byte[] encodedData = Encoding.Unicode.GetBytes(sensitiveData);
```

```
    // 创建文件流和加密服务提供者对象
    using (FileStream fileStream = new FileStream(path,
                                       FileMode.Create,
                                       FileAccess.Write))
    {
        // 生成并保存密钥和初始向量
        GenerateSecretKey();
        GenerateSecretInitVector();

        // 创建加密转换和流对象
        using (ICryptoTransform transform =
                  symmetricAlgorithm.CreateEncryptor(savedKey,
                          savedIV))
        {
            using (CryptoStream cryptoStream =
                   new CryptoStream(fileStream, transform,
                                    CryptoStreamMode.Write))
            {
                // 将加密后的数据写入文件
                cryptoStream.Write(encodedData, 0, encodedData.Length);
            }
        }
    }
}

public string ReadSensitiveData()
{
    string decrypted = "";

    // 创建文件流以回读加密文件
    using (FileStream fileStream = new FileStream(path,
                                       FileMode.Open,
                                       FileAccess.Read))
    {
        // 输出加密文件的内容
        using (BinaryReader binReader = new BinaryReader(fileStream))
        {
            Console.WriteLine("---------- Encrypted Data ---------");
            int count = (Convert.ToInt32(binReader.BaseStream.Length));
            byte [] bytes = binReader.ReadBytes(count);
            char [] array = Encoding.Unicode.GetChars(bytes);
            string encdata = new string(array);
            Console.WriteLine(encdata);
            Console.WriteLine($"---------- Encrypted Data ---------
                               {Environment.NewLine}");

            // 重置文件流
            fileStream.Seek(0,SeekOrigin.Begin);

            // 创建解密器
            using (ICryptoTransform transform =
                symmetricAlgorithm.CreateDecryptor(savedKey, savedIV))
            {
                using (CryptoStream cryptoStream = new CryptoStream(fileStream,
```

```
                                                     transform,
                                                     CryptoStreamMode.Read))
                {
                    // 输出已解密文件的内容
                    using (StreamReader srDecrypted =
                           new StreamReader(cryptoStream, new UnicodeEncoding()))
                    {
                        Console.WriteLine("---------- Decrypted Data ---------");
                        decrypted = srDecrypted.ReadToEnd();
                        Console.WriteLine(decrypted);
                        Console.WriteLine($"---------- Decrypted Data ---------
                                           {Environment.NewLine}");
                    }
                }
            }
        }
    }

    return decrypted;
}

private void GenerateSecretKey()
{
    if (null != (symmetricAlgorithm as TripleDESCryptoServiceProvider))
    {
        TripleDESCryptoServiceProvider tdes;
        tdes = symmetricAlgorithm as TripleDESCryptoServiceProvider;
        tdes.KeySize = 192; // 最大的密钥大小
        tdes.GenerateKey();
        savedKey = tdes.Key;
    }
    else if (null != (symmetricAlgorithm as RijndaelManaged))
    {
        RijndaelManaged rdProvider;
        rdProvider = symmetricAlgorithm as RijndaelManaged;
        rdProvider.KeySize = 256; // 最大的密钥大小
        rdProvider.GenerateKey();
        savedKey = rdProvider.Key;
    }
}

private void GenerateSecretInitVector()
{
    if (null != (symmetricAlgorithm as TripleDESCryptoServiceProvider))
    {
        TripleDESCryptoServiceProvider tdes;
        tdes = symmetricAlgorithm as TripleDESCryptoServiceProvider;
        tdes.GenerateIV();
        savedIV = tdes.IV;
    }
    else if (null != (symmetricAlgorithm as RijndaelManaged))
    {
        RijndaelManaged rdProvider;
        rdProvider = symmetricAlgorithm as RijndaelManaged;
        rdProvider.GenerateIV();
```

```
                savedIV = rdProvider.IV;
            }
        }
```

如果将 SaveSensitiveData 方法用于把下列文本存储到文件中：

```
This is a test
This is sensitive data!
```

那么，ReadSensitiveData 方法将显示来自相同文件的下列信息：

```
---------- Encrypted Data --------
????????????????????????????????????????
---------- Encrypted Data --------

---------- Decrypted Data ---------
This is a test
This is sensitive data!
---------- Decrypted Data ---------
```

11.2.3 讨论

加密数据对很多应用程序都是很必要的，特别是那些将信息存储在易访问位置的应用程序。一旦数据被加密，就需要一种解密方案将数据重新恢复成未加密的形式而不丢失任何信息。

本范例中使用的加密方案是 TripleDES 和 Rijndael。使用 TripleDES 有以下几个原因。

- TripleDES 使用对称加密，意味着使用一个单独的私钥加密和解密数据。这一过程允许更快的加密和解密，特别是当数据流变得更大时。
- TripleDES 加密算法的破解比老的 DES 加密术要难得多，被广泛认为是高强度的。
- 如果你希望另一种类型的加密，那么可使用从 SymmetricAlgorithm 类派生出的任意提供者轻松地转换该范例。
- TripleDES 目前在业内得到了广泛的应用

TripleDES 的主要缺陷在于发送方和接收方必须使用相同的密钥和初始向量（IV）对数据进行成功的加密和解密。如果你希望拥有更安全的加密方案，那么可以使用 Rijndael 方案。该加密方案的类型被认为是一种可靠的加密方案，因为它的运行速度快并且可以使用比 TripleDES 更大的密钥。但是，它仍是一种对称密码系统，这意味着它依赖共享密钥。对于使用共享公钥并且带有不在参与方之间共享的私钥的密码系统，可以使用非对称密码系统，如 RSA 和 DSA。

11.2.4 参考

MSDN 文档中的“SymmetricAlgorithm 类”“TripleDESCryptoServiceProvider 类”和“RijndaelManaged 类”主题。

11.3 清理密码算法信息

11.3.1 问题

你要使用 FCL 中的密码算法类来加密或解密数据。为此，你希望确保数据（例如种子值或密钥）留在内存中的时间不长于正在使用的密码类。黑客有时会在内存中发现该信息，并使用它破译你的加密；更糟糕的是，黑客可能破译用户的密码、修改数据，然后重新加密数据，迫使你的应用程序使用被破坏过的数据而非合法数据。

11.3.2 解决方案

为了清理密钥和初始向量（或种子值），需要调用从你使用的任意 `SymmetricAlgorithm` 或 `AsymmetricAlgorithm` 派生出的类上的 `Clear` 方法。`Clear` 重新初始化密钥和 IV 属性，防止在内存中发现它们。在存储了密钥和 IV 之后调用它，从而可以在随后进行解密。例 11-3 展示了如何加密一个字符串，随后立即清理，尽可能降低潜在攻击者的攻击机会。

例 11-3：清理密码算法信息

```
using System;
using System.Text;
using System.IO;
using System.Security.Cryptography;

public static void CleanUpCrypto()
{
    string originalStr = "SuperSecret information";
    // 将数据字符串编码以存储在内存中
    byte[] originalStrAsBytes = Encoding.ASCII.GetBytes(originalStr);

    // 创建MemoryStream以包含输出
    MemoryStream memStream = new MemoryStream(originalStrAsBytes.Length);
    RijndaelManaged rijndael = new RijndaelManaged();

    // 生成密钥和初始向量
    rijndael.KeySize = 256;
    rijndael.GenerateKey();
    rijndael.GenerateIV();

    // 保存密钥和IV以用于之后的解密
    byte [] key = rijndael.Key;
    byte [] IV = rijndael.IV;

    // 创建加密器和流对象
    ICryptoTransform transform = rijndael.CreateEncryptor(rijndael.Key,
        rijndael.IV);
    CryptoStream cryptoStream = new CryptoStream(memStream, transform,
        CryptoStreamMode.Write);

    // 将加密过的数据写入到MemoryStream
    cryptoStream.Write(originalStrAsBytes, 0, originalStrAsBytes.Length);
    cryptoStream.FlushFinalBlock();
```

```
        // 完成之后立即释放所有资源
        // 以避免在内存中保持任何信息
        memStream.Close();
        cryptoStream.Close();
        transform.Dispose();
        // clear语句重成生成密钥和初始向量
        // 因此留在内存中的不再是你用于加密的信息
        rijndael.Clear();
    }
```

你还可以通过使用 using 语句使工作变得更简单些，不必手动对各 Close 方法进行调用。下列代码块展示了如何使用 using 语句。

```
public static void CleanUpCryptoWithUsing()
{
    string originalStr = "SuperSecret information";
    // 将数据字符串编码以存储在内存中
    byte[] originalStrAsBytes = Encoding.ASCII.GetBytes(originalStr);
    byte[] originalBytes = { };

    // 创建MemoryStream以包含输出
    using (MemoryStream memStream = new MemoryStream(originalStrAsBytes.Length))
    {
        using (RijndaelManaged rijndael = new RijndaelManaged())
        {
            // 生成密钥和初始向量
            rijndael.KeySize = 256;
            rijndael.GenerateKey();
            rijndael.GenerateIV();

            // 保存密钥和IV以用于之后的解密
            byte[] key = rijndael.Key;
            byte[] IV = rijndael.IV;

            // 创建加密器和流对象
            using (ICryptoTransform transform =
                    rijndael.CreateEncryptor(rijndael.Key, rijndael.IV))
            {
                using (CryptoStream cryptoStream = new
                       CryptoStream(memStream, transform,
                        CryptoStreamMode.Write))
                {
                    // 将加密过的数据写入MemoryStream
                    cryptoStream.Write(originalStrAsBytes, 0,
                                originalStrAsBytes.Length);
                    cryptoStream.FlushFinalBlock();
                }
            }
        }
    }
}
```

11.3.3 讨论

为确保数据安全，你需要尽快关闭 MemoryStream 和 CryptoStream 对象，并调用 ICryptoTransform 上的 Dispose，清理加密中使用的所有资源。using 语句令这一过程变得简单，令你的代码更易阅读，并且减少了编程错误。

11.3.4 参考

MSDN 文档中的“SymmetricAlgorithm.Clear 方法”和“AsymmetricAlgorithm.Clear 方法”主题。

11.4 避免字符串在传输或静止时被篡改

11.4.1 问题

你需要将一些文本跨网络发送到另一台机器进行处理或者将其放置在存储介质中以便于以后检索。你需要验证此文本仍未被修改、未被干预，并且未损坏。

11.4.2 解决方案

从字符串计算散列值，将散列值数字化签名，并向接收方同时发送字符串和它的数字签名（公钥也将提供给接收方）。一旦目标接收到此信息，就可以通过验证不能被伪造或修改的数字签名，来确定该字符串是否是最初发送的那一个。

在探究如何工作的细节之前，首先来看一下用于对字符串数据进行数字签名以及反过来用相同的数字签名确认此字符串未发生更改的代码。在例 11-4 中，AntiTamper 类包含两个方法，SignString 和 VerifySignedString，它们分别执行相应的职责。SignString 方法接受明文字符串，并且从它生成数字签名。VerifySignedString 方法由接收到字符串的代码来使用，以确定在接收之前该字符串是否以任何方式修改过。

例 11-4：AntiTamper 类

```
public class AntiTamper
{
    static private readonly int RSA_KEY_SIZE = 2048;


    public static byte[] SignString(string clearText, out string rsaPublicKey)
    {
        byte[] signature = null;
        rsaPublicKey = null;

        byte[] encodedClearText = Encoding.Unicode.GetBytes(clearText);

        using (SHA512CryptoServiceProvider sha512 =
                            new SHA512CryptoServiceProvider())
        {
            using (RSACryptoServiceProvider rsa =
```

```
                new RSACryptoServiceProvider(RSA_KEY_SIZE))
            {
                signature = rsa.SignData(encodedClearText, sha512);

                rsaPublicKey = rsa.ToXmlString(false);
            }
        }

        return signature;
    }

    public static bool VerifySignedString(string clearText, byte[] signature,
                                          string rsaPublicKey)
    {
        bool verified = false;
        byte[] encodedClearText = Encoding.Unicode.GetBytes(clearText);

        using (SHA512CryptoServiceProvider sha512 =
                     new SHA512CryptoServiceProvider())
        {
            using (RSACryptoServiceProvider rsa =
                   new RSACryptoServiceProvider(RSA_KEY_SIZE))
            {
                rsa.FromXmlString(rsaPublicKey);

                verified = rsa.VerifyData(encodedClearText, sha512, signature);
            }
        }

        return verified;
    }
}
```

VerifyStringIntegrity 方法演示如何使用 AntiTamper 类进行签名和验证字符串。VerifyStringIntegrity 方法首先调用 SendData 方法。此方法封装了存在于发送方的代码，但你将需要添加真实地向接收方发送完整消息的代码。发送消息之前，此方法从字符串数据生成我们想要用来保护字符串免受篡改的数字签名。通过调用静态的 AntiTamper.SignString 方法生成数字签名。此方法以 byte[] 返回一个数字签名，并且通过 out 参数返回 RSA 公钥信息。验证方法 ReceiveData 需要此 RSA 公钥信息。

重点是理解接收方需要以下三个项目：原始字符串数据，它的数字签名，以及公钥。字符串数据和签名可以在同一条消息中发送；然而，公钥可以连同消息一起发送，也可以通过单独的通道进行分发。这个单独的通道可以是以下几种机制之一：经过签名和加密的电子邮件，安全的 FTP 服务器，由信任的第三方权威机构签名的 X.509 证书，简单公钥基础架构（SPKI）或者使用 Pretty Good Privacy（PGP）签名和加密公钥以证明其来源于预期中的一方。

无论你使用何种机制来分发公钥，至关重要的是接收方信任此公钥确实来自正确的一方。

第二个方法 ReceiveData 接收字符串数据、生成的数字签名和 RSA 公钥信息，使用数字签名来验证收到的字符串数据。此方法封装了存在于接收方一方的代码，但你将需要添加真实地从发送方接收完整消息的代码。如果数字签名确定证明了字签串数据未被篡改，则返回布尔值 true，否则返回 false，指示该字符串数据已被修改或篡改。

```
public static void VerifyStringIntegrity()
{
    string originalString = "This is the string that we'll be testing.";

    // 从我们需要保护的原始字符串值中创建一个散列值，
    // 并对散列值签名
    string rsaPublicKey;
    byte[] signature = SendData(originalString, out rsaPublicKey);

    // 取消下行代码的注释以快速测试处理一个篡改过的字符串
    //      originalString += "a";
    // 取消下行代码的注释以快速测试处理一个篡改过的签名
    //      signature[1] = 100;

    // 现在,确认字符串未被损坏,未被篡改
    if (ReceiveData(originalString, signature, rsaPublicKey))
    {
        Console.WriteLine(
            "The original string was NOT corrupted or tampered with.");
    }
    else
    {
        Console.WriteLine(
            "ALERT:  The original string was corrupted and/or tampered with.");
    }
}

private static byte[] SendData(string originalString, out string rsaPublicKey)
{
    // 对字符串数据进行数字签名
    byte[] signature = AntiTamper.SignString(originalString, out rsaPublicKey);

    // 将数据发送到目标

    return signature;
}

private static bool ReceiveData(string originalString, byte[] signature,
                                string rsaPublicKey)
{
    // 从发送方接收到数据

    // 验证数字签名
    return (AntiTamper.VerifySignedString(originalString, signature,
        rsaPublicKey));
}
```

当字符串未被破坏时，该方法的输出如下所示。

```
The original string was NOT corrupted or tampered with.
```

当字符串遭到篡改时，该方法的输出如下所示。

```
ALERT:  The original string was corrupted and/or tampered with.
```

若实际查看这一情况，只需要取消 VerifyStringIntegrity 方法中的以下两个注释行之一：

```
// 取消下行代码的注释以快速测试处理一个篡改过的字符串
    originalString += "a";
```

以及：

```
// 取消下行代码的注释以快速测试处理一个篡改过的签名
    signature[1] = 100;
```

11.4.3 讨论

散列值对于确定数据在静止或传输时是否被修改是非常有用的。首先从想要保护的数据中计算出一个散列值［甚至是一个校验和或者循环冗余检查（CRC）值］。然后将此散列值与数据一起发送到接收方。接收方基于收到的数据重新计算散列值。如果新的散列值与收到的散列值匹配，则数据未被更改；否则说明数据在某种程度上已被修改或损坏。

双方协定使用同一个散列算法是至关重要的。SHA-256 和 SHA-512 算法都是一个不错的安全选择，同时也是行业标准。

虽然这个散列技术在标识数据已被损坏或被意外修改方面工作良好，但它无法防止攻击者偷偷尝试修改数据以获得对系统的访问或者散布假信息以企图敲诈或勒索。如果只有一个散列值用来保护数据，攻击者可以截获数据（使用中间人攻击），修改数据，然后使用修改后的数据重新生成一个新的散列。之后在发送到预期接收方之前使用新的散列替换旧的散列值。接收方无法得知数据已被篡改；因为从接收方的角度来看，接收方生成的散列值与接收到的散列值是相同的。若要防止这些类型的攻击，需要更健壮的系统。这就是数字签名发挥作用之处。

数字签名是通过一种非对称公钥加密算法生成的。这意味着有两把密钥。第一把是可以分发给所有将会收到已签名数据的协作方的公钥。此公钥将用于验证所接收的数据的数字签名。第二把是必须安全地存放在数据发送方的私钥。私钥仅用于将数据发送给接收方之前数据进行初始签名。公钥和私钥一起工作，一个用来对数据进行签名，另一个不仅证明签名来自预期发送方，也保证使用该签名签名过的数据未被篡改、修改或损坏。

如果私钥被盗，攻击者将能够对数据进行数字签名，假装他是数据的合法发送方。永远不要将私钥发送给不必要的一方，并且永远不要以明文发送或存储私钥。

下面介绍数据是如何由发送方进行数字签名的。AntiTamper.SignString 方法被调用，要进行签名的数据被传入到第一个参数（clearText）中，一个字符串变量（rsaPublicKey）是作为第二个参数传入的。rsaPublicKey 变量最终将包含公钥信息，该信息必须用于随后在 AntiTamper.VerifySignedString 方法中验证签名。

```
public static byte[] SignString(string clearText, out string rsaPublicKey)
{
    byte[] signature = null;
    rsaPublicKey = null;

    byte[] encodedClearText = Encoding.Unicode.GetBytes(clearText);

    using (SHA512CryptoServiceProvider sha512 =
                     new SHA512CryptoServiceProvider())
    {
        using (RSACryptoServiceProvider rsa =
                 new RSACryptoServiceProvider(RSA_KEY_SIZE))
        {
            signature = rsa.SignData(encodedClearText, sha512);

            rsaPublicKey = rsa.ToXmlString(false);
        }
    }

    return signature;
}
```

首先，SignString 方法创建一个 SHA512CryptoServiceProvider 对象，它将用于创建将会被数字签名的散列。此处要注意到，我们为需要保护的数据创建一个 SHA-512 散列值。然而，我们并没有对要保护的数据进行签名，而是对 SHA-512 散列值签名。这十分重要，因为非对称密码算法本身是很慢的。如果我们对保护的数据进行签名，并且这些数据非常大（例如，大小达到数 MB 或数 GB），签名过程将会拖慢整个系统。通过签署小得多的散列值（在本例中为 512 字节），我们不需要担心性能瓶颈。

接下来，创建一个将会用来对数据进行签名的 RSACryptoServiceProvider 对象。RSACryptoServiceProvider.SignData 实例方法接受要进行签名的 byte[] 形式的明文数据，以及我们要使用的散列算法（SHA-512）。这些用来生成散列值，然后再生成数字签名。此方法仅返回数字签名。

最后还有一个非常重要的步骤，即捕获 RSACryptoServiceProvider 对象所生成的公钥信息。我们通过调用 RSACryptoServiceProvider.ToXmlString 实例方法完成这一步。此方法返回验证签名时需要的公钥信息。

> 当调用 ToXMLString 时，传入布尔值 false 以仅返回公钥。如果你传入 true，则会同时返回公钥和私钥。如前所述，对密钥进行保护且不意外地分发是非常必要的。

现在发送方所要做的就是将数据、数字签名和 AntiTamper.SignString 方法返回的公钥信息发送给预期的接收方，代码如下所示。

```
private static byte[] SendData(string originalString, out string rsaPublicKey)
{
    // 将数据发送到目标
    byte[] signature = AntiTamper.SignString(originalString, out rsaPublicKey);

    // 将数据和签名发送到目标

    return signature;
}
```

接收方然后调用 AntiTamper.VerifySignedString 方法，传入接收到的数据、数字签名和公钥信息。注意，AntiTamper 类在发送方和接收方的代码中都需要引用到。

```
private static bool ReceiveData(string originalString, byte[] signature,
                                string rsaPublicKey)
{
    // 从发送方接收到数据

    // 验证数字签名
    return (AntiTamper.VerifySignedString(originalString, signature,
        rsaPublicKey));
}
```

VerifySignedString 方法必须使用发送方在之前的 SignString 方法中使用过的相同 SHA512CryptoServiceProvider 对象；否则，签名将无法验证。此外创建了一个 RSACryptoServiceProvider 对象，但使用此对象来验证签名之前，调用 RSACryptoService-Provider.FromXmlString 方法来导入正确验证签名所需的公钥信息。最后，RSACryptoServiceProvider.VerifyData 方法被调用以验证数据及其签名。如果字符串数据未被篡改或损坏，此方法返回布尔值 true，否则返回 false。

```
public static bool VerifySignedString(string clearText, byte[] signature,
                                      string rsaPublicKey)
{
    bool verified = false;
    byte[] encodedClearText = Encoding.Unicode.GetBytes(clearText);

    using (SHA512CryptoServiceProvider sha512 =
                     new SHA512CryptoServiceProvider())
    {
        using (RSACryptoServiceProvider rsa =
                new RSACryptoServiceProvider(RSA_KEY_SIZE))
        {
            rsa.FromXmlString(rsaPublicKey);

            verified = rsa.VerifyData(encodedClearText, sha512, signature);
        }
    }

    return verified;
}
```

11.4.4 参考

MSDN 文档中的“RSACryptoServiceProvider 类”“SHA512CryptoServiceProvider 类”和“Encoding.Unicode.GetBytes 方法”主题。关于公钥的更多信息，参考维基百科的“Public-key cryptography”。

11.5 保证安全断言的安全

11.5.1 问题

你希望断言在调用栈的某个特定点上，一个给定的权限对所有后续调用都可用。但是，这样做很容易打开一个安全漏洞，允许其他恶意代码哄骗你的代码或者在你的组件中创建一个后门。你希望断言一个给定的安全权限，但希望以一种安全、高效的方式进行。

11.5.2 解决方案

为了确保该方法的安全，你需要调用后续调用所需的权限上的 Demand。这会确保那些不具有这些权限的代码不会因为 Assert 而蒙混过关。Demand 可确保你在使用对栈短路的 Assert 之前确实已经获得了这一许可。这通过 CallSecureFunctionSafelyAndEfficiently 函数举例进行说明，它在调用 SecureFunction 之前，执行一个 Demand 和一个 Assert，SecureFunction 相应地对 ReflectionPermission 执行一个 Demand。

CallSecureFunctionSafelyAndEfficiently 的代码如例 11-5 所示。

例 11-5：CallSecureFunctionSafelyAndEfficiently 函数

```
public static void CallSecureFunctionSafelyAndEfficiently()
{

    // 建立一个权限以便能够通过反射访问非公开成员
    ReflectionPermission perm =
        new ReflectionPermission(ReflectionPermissionFlag.MemberAccess);

    // 在使用Assert前请求编制的权限集
    // 以确认在Assert前拥有相应的权限
    // Demand可确保我们在使用对栈短路的Assert之前
    // 已经检查了这一权限,这有助有我们保持安全,并更好地运行
    perm.Demand();

    // 在调用到同样执行Demand以短路每个
    // 调用生成的栈之前断言此权限
    // 断言帮助我们优化对SecureFunction的使用
    perm.Assert();

    // 我们调用安全函数100次,但仅生成从该函数到当前调用函数的堆栈审核,
    // 而不是审核完整的堆栈100次
    for(int i=0;i<100;i++)
    {
        SecureFunction();
```

```
        }
    }
```

SecureFunction 的代码如下所示。

```
    public static void SecureFunction()
    {
        // 建立一个权限以便能够通过反射访问非公开成员
        ReflectionPermission perm =
            new ReflectionPermission(ReflectionPermissionFlag.MemberAccess);

        // 请求执行此操作的权限,导致一个堆栈审核
        perm.Demand();

        // 此处执行动作
    }
```

11.5.3 讨论

在示例函数 CallSecureFunctionSafelyAndEfficiently 中，你调用的函数（SecureFunction）执行 ReflectionPermission 上的一个 Demand，确保代码能够通过反射访问非公开的类成员。一般而言，对于 SecureFunction 的每次调用，都会导致一次栈审核。CallSecureFunctionSafelyAndEfficiently 中的 Demand 只保护第一个位置中 Assert 的使用。为了使之效率更高，你可以使用 Assert，声明引发被它调用的 Demand 的所有函数不会再引发栈审核。Assert 表示停止对调用栈中的该权限进行检查。为了完成这项工作，你需要调用 Assert 的权限。

使用 Assert 会引发问题，因为它在通过 CallSecureFunctionSafelyAndEfficiently 调用 SecureFunction 的地方打开一个潜在的引诱攻击，它调用 Assert 阻止 Demand 引发 SecureFunction 的栈审核。假使未带 ReflectionPermission 的未授权代码能够调用 CallSecureFunctionSafelyAndEfficiently，那么 Assert 将会阻止 SecureFunction 的 Demand 调用，而此 Demand 调用是用来确定调用栈中是否存在某些代码不具有适当的权限。这就是当一个 Demand 发生时 CLR 中调用栈检查的能力。

为了防止问题的发生，你要在执行 Assert 之前对 CallSecureFunctionSafelyAndEfficiently 中的 SecureFunction 所需的 ReflectionPermission 执行一个 Demand，以关闭这个漏洞。这个 Demand 和 Assert 的联合使用会导致一次栈审核，而不是最初由 SecureFunction 中的 Demand 所引发的 100 次。

安全性优化技术，例如在这种情况下使用 Assert（即使它不是使用 Assert 的主要原因），可帮助类库和受信任的控件开发人员执行 Assert，从而加速其代码与 CLR 的交互；但如果使用不当，这些技术也可能打开安全规划中的漏洞。这个示例说明在涉及安全访问时你可以同时兼顾性能和安全性。

如果你正在使用 Assert，那么请留心不要在类构造函数中进行栈审核重写。构造函数不能保证拥有任何特定的安全上下文，也不能保证在特定时间点上执行。因此，调用栈无法很好地进行定义，而这里使用的 Assert 可能产生意想不到的结果。

使用 Assert 要记住的另一件事是在某一时间内一个函数中只能有一个激活的 Assert。如

果对相同的许可 Assert 两次，那么 CLR 将会引发一个 SecurityException。你必须首先使用 RevertAssert 撤消最初的 Assert，然后再声明第二个 Assert。

11.5.4 参考

MSDN 文档中的“CodeAccessSecurity.Assert 方法”“CodeAccessSecurity.Demand 方法”“CodeAccessSecurity.RevertAssert 方法”和“Overriding Security Checks”主题。

11.6 验证是否已授予程序集特定权限

11.6.1 问题

当你的程序集使用 SecurityAction.RequestOptional 标志要求可选的权限（例如，要求磁盘访问使用户将数据导出到磁盘作为一个产品特性）时，它可能获得那些权限，也可能无法获得。不管怎样，你的程序集仍能加载并执行。你需要一种方法去验证程序集是否确实得到了那些权限。这有助于防止许多安全性异常的引发。例如，如果你选择性地要求注册表上的读 / 写权限但没有获得它们，那么可以禁用用于读取和存储注册表中应用程序设置的用户界面控件。

11.6.2 解决方案

如下使用 SecurityManager.IsGranted 方法检查你的程序集是否获得了可选的权限。

```
using System;
using System.Text.RegularExpressions;
using System.Web;
using System.Net;
using System.Security;

Regex regex = new Regex(@"http://www\.oreilly\.com/.*");
WebPermission webConnectPerm = new WebPermission(NetworkAccess. Connect,regex);

PermissionSet pSet = new PermissionSet(PermissionState.None);
pSet.AddPermission(webConnectPerm);
if (pSet.IsSubsetOf(Assembly.GetExecutingAssembly().PermissionSet))
{
   // 连接到O’Reilly网站
}
```

该代码为 O’Reilly 的网站建立一个 Regex，然后使用它创建一个 WebPermission 用于连接该站点和包含该字符串的所有站点。然后通过创建一个新的不包含访问受保护资源权限（即 PermissionState.None）的 PermissionSet 对象，将 webConnectPerm 权限添加到新创建的 PermissionSet 对象，并且最后检查这一新的 PermissionSet 对象是否为执行中的程序集的权限集的子集来检查 WebPermission。

11.6.3 讨论

IsSubsetOf 方法是一个轻量级方法，用于确定权限是否被赋予某个程序集，无需一个 Demand 调用首先带来的完全栈审核。但要注意，一旦你运行了执行 Demand 的代码，那么将会引发完全栈审核。

你需要设计具备可选权限的程序集的一个原因是为了适应不同客户环境中的部署。在某些情况下（例如桌面应用程序），拥有一个能够执行更健壮动作（与数据库会话，创建网络通信，等等）的程序集是可接受的。在其他情况下，如果客户不希望为这些额外服务的运行赋予足够的权限，那么你可以推迟这些动作。

11.6.4 参考

MSDN 文档中的“WebPermission 类”“SecurityManager 类”和“IsSubsetOf 方法”主题。

11.7 最小化程序集的攻击面

11.7.1 问题

攻击你的程序集的某个人首先会试图尽量多地找出关于你的程序集的信息，然后在构建攻击时使用这些信息。你为攻击者留出的区域越多，他们能够使用的就越多。你需要将自己的程序集所允许做的事最小化。这样，如果一个攻击者成功接管它，那么攻击者将无法获得对系统造成任何损失的必要权限。

11.7.2 解决方案

使用 SecurityAction.RequestRefuse 枚举成员在程序集级别上指定你不希望该程序集拥有的权限。这将强制 CLR 拒绝把这些权限授予你的代码，并且确保即使系统的另一部分受到威胁，你的代码也不会用于执行不需要权限的功能。

下列示例允许程序集执行文件 I/O 作为其最小许可集的一部分，但显式地拒绝允许该程序集拥有权限以跳过验证：

```
[assembly: FileIOPermission(SecurityAction.RequestMinimum,Unrestricted=true)]
[assembly: SecurityPermission(SecurityAction.RequestRefuse,
           SkipVerification=false)]
```

11.7.3 讨论

一旦你决定了你的程序集需要哪些权限作为其普通安全测试的一部分，那么就可以使用 RequestRefuse 锁定代码。如果这看起来比较极端，那么可考虑你的代码能够访问一个包含敏感信息的数据存储的情况，例如社保号或工资信息。这一积极步骤有助于你向客户展示自己对待安全性的认真态度，并且当入侵发生在把你的代码作为其一部分的系统上时有助于保护你的利益。

对该方法需要慎重考虑的是，使用 RequestRefuse 会将你的程序集标记为部分受信任的。这相应地阻止了它调用任何未使用 AllowPartiallyTrustedCallers 属性标记的强命名程序集。

11.7.4　参考

MSDN 文档中的“通过部分受信任的代码使用库”“SecurityAction 枚举”和“全局特性”主题。

11.8　获得安全和/或审计信息

11.8.1　问题

你需要获得一个文件或注册表项的安全权限和 / 或审计信息。

11.8.2　解决方案

当要获取一个文件的安全和 / 或审计信息时，可使用 File 类的静态 GetAccessControl 方法，以获得一个 System.Security.AccessControl.FileSecurity 对象。使用 FileSecurity 对象访问文件的安全和审计信息。这些步骤如例 11-6 所示。

例 11-6：获得安全审计信息

```
public static void ViewFileRights()
{
    // 获得一个文件的安全信息
    string file = @"C:\Windows\win.ini";
    FileSecurity fileSec = File.GetAccessControl(file);
    DisplayFileSecurityInfo(fileSec);
}

public static void DisplayFileSecurityInfo(FileSecurity fileSec)
{
    Console.WriteLine($"GetSecurityDescriptorSddlForm:
            {fileSec.GetSecurityDescriptorSddlForm(AccessControlSections.All)}");

    foreach (FileSystemAccessRule ace in
          fileSec.GetAccessRules(true, true, typeof(NTAccount)))
    {
        Console.WriteLine("\tIdentityReference.Value:
                     {ace.IdentityReference.Value}");
        Console.WriteLine($"\tAccessControlType: {ace.AccessControlType}");
        Console.WriteLine($"\tFileSystemRights: {ace.FileSystemRights}");
        Console.WriteLine($"\tInheritanceFlags: {ace.InheritanceFlags}");
        Console.WriteLine($"\tIsInherited: {ace.IsInherited}");
        Console.WriteLine($"\tPropagationFlags: {ace.PropagationFlags}");

        Console.WriteLine("-----------------\r\n\r\n");
    }
```

```
    foreach (FileSystemAuditRule ace in
            fileSec.GetAuditRules(true, true, typeof(NTAccount)))
    {
        Console.WriteLine("\tIdentityReference.Value:
                           {ace.IdentityReference.Value}");
        Console.WriteLine($"\tAuditFlags: {ace.AuditFlags}");
        Console.WriteLine($"\tFileSystemRights: {ace.FileSystemRights}");
        Console.WriteLine($"\tInheritanceFlags: {ace.InheritanceFlags}");
        Console.WriteLine($"\tIsInherited: {ace.IsInherited}");
        Console.WriteLine($"\tPropagationFlags: {ace.PropagationFlags}");

        Console.WriteLine("-----------------\r\n\r\n");
    }

    Console.WriteLine($"GetGroup(typeof(NTAccount)).Value:
                        {fileSec.GetGroup(typeof(NTAccount)).Value}");
    Console.WriteLine($"GetOwner(typeof(NTAccount)).Value:
                        {fileSec.GetOwner(typeof(NTAccount)).Value}");

    Console.WriteLine("---------------------------------------\r\n\r\n\r\n");
}
```

这些方法产生下列输出。

```
GetSecurityDescriptorSddlForm:  O:SYG:SYD:AI(A;ID;FA;;;SY)(A;ID;FA;;;BA)
                     (A;ID;0x1200a9;;;BU)(A;ID;0x1200a9;;;AC)
        IdentityReference.Value: NT AUTHORITY\SYSTEM
        AccessControlType: Allow
        FileSystemRights: FullControl
        InheritanceFlags: None
        IsInherited: True
        PropagationFlags: None
-----------------


        IdentityReference.Value: BUILTIN\Administrators
        AccessControlType: Allow
        FileSystemRights: FullControl
        InheritanceFlags: None
        IsInherited: True
        PropagationFlags: None
-----------------


        IdentityReference.Value: BUILTIN\Users
        AccessControlType: Allow
        FileSystemRights: ReadAndExecute, Synchronize
        InheritanceFlags: None
        IsInherited: True
        PropagationFlags: None
-----------------


        IdentityReference.Value:
                         APPLICATION PACKAGE AUTHORITY\ALL APPLICATION PACKAGES
```

```
        AccessControlType: Allow
        FileSystemRights: ReadAndExecute, Synchronize
        InheritanceFlags: None
        IsInherited: True
        PropagationFlags: None
-----------------


GetGroup(typeof(NTAccount)).Value: NT AUTHORITY\SYSTEM
GetOwner(typeof(NTAccount)).Value: NT AUTHORITY\SYSTEM
```

当要获取一个注册表项的安全和 / 或审计信息时，可使用 Microsoft.Win32.RegistryKey 类的 GetAccessControl 方法，获得一个 System.Security.AccessControl.RegistrySecurity 对象。使用 RegistrySecurity 对象访问该注册表项的安全和审计信息。这些步骤如例 11-7 所示。

例 11-7：获得一个注册表项的安全或审计信息

```
public static void ViewRegKeyRights()
{
    // 获得一个注册表项的安全信息
    using (RegistryKey regKey =
        Registry.CurrentUser.OpenSubKey(@"Software\Microsoft\VisualStudio\14.0"))
    {
        RegistrySecurity regSecurity = regKey.GetAccessControl();
        DisplayRegKeySecurityInfo(regSecurity);
    }
}

public static void DisplayRegKeySecurityInfo(RegistrySecurity regSec)
{
    Console.WriteLine($"GetSecurityDescriptorSddlForm:
            {fileSec.GetSecurityDescriptorSddlForm(AccessControlSections.All)}");

    foreach (RegistryAccessRule ace in
            regSec.GetAccessRules(true, true, typeof(NTAccount)))
    {
        Console.WriteLine("\tIdentityReference.Value:
                           {ace.IdentityReference.Value}");
        Console.WriteLine($"\tAccessControlType: {ace.AccessControlType}");
        Console.WriteLine($"\tFileSystemRights: {ace.FileSystemRights}");
        Console.WriteLine($"\tInheritanceFlags: {ace.InheritanceFlags}");
        Console.WriteLine($"\tIsInherited: {ace.IsInherited}");
        Console.WriteLine($"\tPropagationFlags: {ace.PropagationFlags}");

        Console.WriteLine("-----------------\r\n\r\n");
    }

    foreach (RegistryAuditRule ace in
            regSec.GetAuditRules(true, true, typeof(NTAccount)))
    {
        Console.WriteLine("\tIdentityReference.Value:
                           {ace.IdentityReference.Value}");
        Console.WriteLine($"\tAuditFlags: {ace.AuditFlags}");
        Console.WriteLine($"\tFileSystemRights: {ace.FileSystemRights}");
```

```
        Console.WriteLine($"\tInheritanceFlags: {ace.InheritanceFlags}");
        Console.WriteLine($"\tIsInherited: {ace.IsInherited}");
        Console.WriteLine($"\tPropagationFlags: {ace.PropagationFlags}");

        Console.WriteLine("-----------------\r\n\r\n");
    }
    Console.WriteLine($"GetGroup(typeof(NTAccount)).Value:
                    {fileSec.GetGroup(typeof(NTAccount)).Value}");
    Console.WriteLine($"GetOwner(typeof(NTAccount)).Value:
                    {fileSec.GetOwner(typeof(NTAccount)).Value}");

    Console.WriteLine("---------------------------------------\r\n\r\n\r\n");
}
```

这些方法产生下列输出。

```
GetSecurityDescriptorSddlForm:  O:S-1-5-21-3613598369-3284219489-1294304910-1001G:
             S-1-5-21-3613598369-3284219489-1294304910-1001D:
            (A;OICIID;KA;;;S-1-5-21-3613598369-3284219489-1294304910-1001)
            (A;OICIID;KA;;;SY)(A;OICIID;KA;;;BA)(A;OICIID;KR;;;RC)
        IdentityReference.Value: VM_Win81_VS14\Teilhet
        AccessControlType: Allow
        RegistryRights: FullControl
        InheritanceFlags: ContainerInherit, ObjectInherit
        IsInherited: True
        PropagationFlags: None
-----------------


        IdentityReference.Value: NT AUTHORITY\SYSTEM
        AccessControlType: Allow
        RegistryRights: FullControl
        InheritanceFlags: ContainerInherit, ObjectInherit
        IsInherited: True
        PropagationFlags: None
-----------------


        IdentityReference.Value: BUILTIN\Administrators
        AccessControlType: Allow
        RegistryRights: FullControl
        InheritanceFlags: ContainerInherit, ObjectInherit
        IsInherited: True
        PropagationFlags: None
-----------------


        IdentityReference.Value: NT AUTHORITY\RESTRICTED
        AccessControlType: Allow
        RegistryRights: ReadKey
        InheritanceFlags: ContainerInherit, ObjectInherit
        IsInherited: True
        PropagationFlags: None
-----------------
```

```
GetGroup(typeof(NTAccount)).Value: VM_WIN81_VS14\Teilhet
GetOwner(typeof(NTAccount)).Value: VM_WIN81_VS14\Teilhet
```

11.8.3 讨论

用于获得文件或注册表项的安全信息的基本方法是 `GetAccessControl` 方法。当在 `RegistryKey` 对象上调用该方法时，会返回一个 `RegistrySecurity` 对象。但是，当在 `File` 类上调用该方法时，会返回一个 `FileSecurity` 对象。`RegistrySecurity` 和 `FileSecurity` 对象本质上代表一个*自由访问控制表*（discretionary access control list，DACL）。非托管语言（例如 C++）的开发人员习惯使用 DACL。

`RegistrySecurity` 和 `FileSecurity` 对象都包含一个安全规则列表，应用于它代表的系统对象。`RegistrySecurity` 对象包含一个 `RegistryAccessRule` 对象的列表，而 `FileSecurity` 对象包含一个 `FileSystemAccessRule` 对象的列表。这些规则对象等价于*访问控制项*（access control entry，ACE），构成了一个 DACL 中安全规则的列表。

除了 `File` 类和 `RegistryKey` 对象之外的系统对象允许查询安全权限。表 11-1 列出了所有返回一个安全对象类型的 .NET Framework 类以及安全对象的类型。此外，还列出了安全对象中包含的规则对象类型。

表11-1：所有*Security和*AccessRule对象及其应用的类型列表

类	GetAccessControl方法返回的对象	安全对象中包含的规则对象类型
Directory	DirectorySecurity	FileSystemAccessRule
DirectoryInfo	DirectorySecurity	FileSystemAccessRule
EventWaitHandle	EventWaitHandleSecurity	EventWaitHandleAccessRule
File	FileSecurity	FileSystemAccessRule
FileInfo	FileSecurity	FileSystemAccessRule
FileStream	FileSecurity	FileSystemAccessRule
Mutex	MutexSecurity	MutexAccessRule
RegistryKey	RegistrySecurity	RegistryAccessRule
Semaphore	SemaphoreSecurity	SemaphoreAccessRule

通过 `*Security` 对象对一个系统对象的 DACL 进行抽象和通过 `*AccessRule` 对象对 DACL 的 ACE 进行抽象，使得对系统对象安全权限的访问变得非常容易。在 .NET Framework 之前的版本中，这些 DACL 及其 ACE 仅允许非托管代码访问。对于 .NET 2.0 Framework 及之后的版本，你能够查看并对这些对象进行编程。

11.8.4 参考

范例 11.9（即 11.9 节）；MSDN 文档中的“`System.IO.File.GetAccessControl` 方法”“`System.Security.AccessControl.FileSecurity` 类”“`Microsoft.Win32.RegistryKey.GetAccessRules` 方法”和“`System.Security.AccessControl.RegistrySecurity` 类”主题。

11.9 授权或撤销对文件或注册表项的访问

11.9.1 问题

你需要以编程方式修改某个文件或注册表项的安全权限。

11.9.2 解决方案

例 11-8 所示的代码授予在一个注册表项上执行写操作的权限，然后撤销。

例 11-8：授予和撤销在一个注册表项上执行写操作的权限

```
public static void GrantRevokeRegKeyRights()
{
    NTAccount user = new NTAccount(@"WRKSTN\ST");

    using (RegistryKey regKey = Registry.LocalMachine.OpenSubKey(
                            @"SOFTWARE\MyCompany\MyApp"))
    {
        GrantRegKeyRights(regKey, user, RegistryRights.WriteKey,
                       InheritanceFlags.None, PropagationFlags.None,
                       AccessControlType.Allow);

        RevokeRegKeyRights(regKey, user, RegistryRights.WriteKey,
                       InheritanceFlags.None, PropagationFlags.None,
                       AccessControlType.Allow)
    }
}

public static void GrantRegKeyRights(RegistryKey regKey,
                                     NTAccount user,
                                     RegistryRights rightsFlags,
                                     InheritanceFlags inherFlags,
                                     PropagationFlags propFlags,
                                     AccessControlType actFlags)
{
    Registry Security regSecurity = regKey.GetAccessControl();

    RegistryAccessRule rule = new RegistryAccessRule(user, rightsFlags, inherFlags,
                                                     propFlags, actFlags);
    regSecurity.AddAccessRule(rule);
    regKey.SetAccessControl(regSecurity);
}

public static void RevokeRegKeyRights(RegistryKey regKey,
                                      NTAccount user,
                                      RegistryRights rightsFlags,
                                      InheritanceFlags inherFlags,
                                      PropagationFlags propFlags,
                                      AccessControlType actFlags)
{
    RegistrySecurity regSecurity = regKey.GetAccessControl();
```

```
        RegistryAccessRule rule = new RegistryAccessRule(user, rightsFlags, inherFlags,
                                                         propFlags, actFlags);
        regSecurity.RemoveAccessRuleSpecific(rule);

        regKey.SetAccessControl(regSecurity);
    }
```

例 11-9 中所示的代码授予删除一个文件的权限，然后撤销。

例 11-9：授予和撤销删除一个文件的权限

```
    public static void GrantRevokeFileRights()
    {
        NTAccount user = new NTAccount(@"WRKSTN\ST");

        string file = @"c:\FOO.TXT";
        GrantFileRights(file, user, FileSystemRights.Delete, InheritanceFlags.None,
                        PropagationFlags.None, AccessControlType.Allow);

        RevokeFileRights(file, user, FileSystemRights.Delete, InheritanceFlags.None,
                         PropagationFlags.None, AccessControlType.Allow);
    }

    public static void GrantFileRights(string file,
                                       NTAccount user,
                                       FileSystemRights rightsFlags,
                                       InheritanceFlags inherFlags,
                                       PropagationFlags propFlags,
                                       AccessControlType actFlags)
    {
        FileSecurity fileSecurity = File.GetAccessControl(file);
        FileSystemAccessRule rule = new FileSystem AccessRule(user, rightsFlags,
                                                              inherFlags, propFlags,
                                                              actFlags);
        fileSecurity.AddAccessRule(rule);
        File.SetAccessControl(file, fileSecurity);
    }

    public static void RevokeFileRights(string file,
                                        NTAccount user,
                                        FileSystemRights rightsFlags,
                                        InheritanceFlags inherFlags,
                                        PropagationFlags propFlags,
                                        AccessControlType actFlags)
    {
        FileSecurity fileSecurity = File.GetAccessControl(file);

        FileSystemAccessRule rule = new FileSystemAccessRule(user, rightsFlags,
                                                             inherFlags, propFlags,
                                                             actFlags);
        fileSecurity.RemoveAccessRuleSpecific(rule);
        File.SetAccessControl(file, fileSecurity);
    }
```

11.9.3 讨论

当授予或撤销对文件或注册表项的访问权限时，你需要两个对象。第一个是合法的 NTAccount 对象。该对象本质上封装了一个用户或组账户，需要用来创建一个新的 RegistryAccessRule 或一个新的 FileSystemAccessRule。NTAccount 确定该访问规则将应用于哪个用户或组。要注意，必须将传递给 NTAccount 构造函数的字符串修改为一个在你机器中存在的合法用户名或组名。如果传递一个已禁用的现存用户或组账户，将会引发一个带有消息“某些或全部身份引用无法被转换”的 IdentityNotMappedException。

所需的第二项是一个合法的 RegistryKey 对象（如果你要修改对注册表项的安全访问），或者是一个包含指向现有文件的合法路径和文件名的字符串。这些对象将具有授权给它们或者从它们中撤销的安全性权限。

一旦获得了这两项，就可以使用第二项获得一个安全对象，它包含访问－规则对象列表。例如，下列代码获得注册表项 HKEY-LOCAL_MACHINE\SOFTWARE\MyCompany\MyApp 的安全对象。

```
RegistryKey regKey = Registry.LocalMachine.OpenSubKey(
                            @"SOFTWARE\MyCompany\MyApp");
RegistrySecurity regSecurity = regKey.GetAccessControl();
```

下列代码获得 FOO.TXT 文件的安全对象。

```
string file = @"c:\FOO.TXT";
FileSecurity fileSecurity = File.Get AccessControl(file);
```

既然你拥有了特定的安全对象，就可以创建一个随后需要添加到该安全性对象上的访问－规则对象。为此，你需要创建一个新的访问规则。对于一个注册表项，必须创建一个新的 RegistryAccessRule 对象；对于一个文件，必须创建一个新的 FileSystemAccessRule 对象。为了将该访问规则添加到正确的安全性对象中，只需调用安全对象上的 SetAccessControl 方法即可。要注意，RegistryAccessRule 对象只能添加到 RegistrySecurity 上，而 FileSystemAccessRule 对象只能够添加到 FileSecurity 对象上。

要从系统对象中删除一个访问－规则对象，可遵循相同的步骤，只是要调用 RemoveAccessRuleSpecific 方法而不是 AddAccessRule 方法。RemoveAccessRuleSpecific 接受一个访问－规则对象，并试图从安全性对象中删除完全匹配该规则对象的规则。与往常一样，你必须记住要调用 SetAccessControl 方法，以将所有修改应用于实际的系统对象。

对于其他允许通过编程修改安全性权限的类型列表，请参考范例 11.8（即 11.8 节）。

11.9.4 参考

范例 11.8（即 11.8 节）；MSDN 文档中的“System.IO.File.GetAccessControl 方法”“System.Security.AccessControl.FileSecurity 类”“Sysem.Security.AccessControl.FileSystemAccessRule 类”“Microsoft.Win32.RegistryKey.GetAccessControl 方法”“System.Security.AccessControl.RegistrySecurity 类”和“System.Security.AccessControl.RegistryAccessRule 类”主题。

11.10 使用安全字符串保护字符串数据

11.10.1 问题

你需要在字符串中存储敏感信息，如社会安全号。但是，你不希望有人在内存中窥视到该数据。

11.10.2 解决方案

使用 SecureString 对象。

要将一个流对象中的文本复制到一个 SecureString 对象中，可使用下列方法。

```
public static SecureString CreateSecureString(StreamReader secretStream)
{
    SecureString secretStr = new SecureString();
    char buf;
    while (secretStream.Peek() >= 0)
    {
        buf = (char)secretStream.Read();
        secretStr.AppendChar(buf);
    }

    // 将secretStr对象标记为只读
    secretStr.MakeReadOnly();
    return (secretStr);
}
```

要从包含敏感数据的字符串中复制文本，可使用下列方法。

```
public static SecureString CreateSecureString(string secret)
{
    SecureString secretStr = new SecureString();
    char[] buf = new char[1];
    foreach (char c in secret)
    {
        secretStr.AppendChar(c);
    }

    // 将secretStr对象标记为只读
    secretStr.MakeReadOnly();

    return (secretStr);
}
```

要从一个 SecureString 对象中取出明文文本，可使用下列方法。

```
public static void ReadSecureString(SecureString secretStr)
{
    // 为了读回字符串,你需要使用一些特殊方法
    IntPtr secretStrPtr = Marshal.SecureStringToBSTR(secretStr);
    string nonSecureStr = Marshal.PtrToStringBSTR(secretStrPtr);
```

```
        // 使用未保护的字符串
        Console.WriteLine($"nonSecureStr = {nonSecureStr}");

        Marshal.ZeroFreeBSTR(secretStrPtr);

        if (!secretStr.IsReadOnly())
        {
            secretStr.Clear();
        }
    }
```

11.10.3 讨论

SecureString 对象特别设计用于包含你希望保密的字符串数据。你可能希望存储在 SecureString 对象中的数据有社会安全号、信用卡号、PIN 码、密码、雇员 ID，或者其他类型的敏感信息。

添加到 SecureString 对象上的字符串数据会立即自动进行加密，当从 SecureString 对象中提取字符串数据时会自动进行解密。加密是使用这个对象的亮点之一。

SecureString 对象的另一特性是当调用了 MakeReadOnly 方法后，SecureString 将成为不可变的。任何修改只读 SecureString 对象内字符串数据的企图都会导致引发一个 InvalidOperationException。一旦 SecureString 对象被置为只读，那么它将无法回到读 / 写状态。尽管如此，在调用一个现有的 SecureString 对象上的 Copy 方法时需要小心。该方法将创建被调用的 SecureString 对象的一个新实例，并带有其数据的副本。但是，这个新 SecureString 对象现在是可读写的。你应当审查自己的代码，以确定是否应将这个新 SecureString 对象与其原始 SecureString 对象一样设置为只读。

> SecureString 对象只能用于 Windows 2000（带 Service Pack 3 或更高版本）或之后的操作系统。

在本范例中，通过从一个流中读取的数据或一个简单字符串创建一个 SecureString 对象。这个数据还可以使用非安全代码从一个 char* 获得。SecureString 对象包含一个构造函数，接受一个 char* 类型的参数以及一个包含长度值的整型参数，它决定了从 char* 中获取的字符数。

从 SecureString 对象中获得数据初看上去并非显而易见。没有可以返回一个 SecureString 对象中包含的数据的方法。为了实现这一点，必须使用 Marshal 类上的两个静态方法。第一个是 SecureStringToBSTR，它接受 SecureString 对象并返回一个 IntPtr。这个 IntPtr 随后被传递给同样位于 Marshal 类上的 PtrToStringBSTR 方法。PtrToStringBSTR 方法然后返回一个包含解密的字符串数据的非安全 String 对象。

一旦使用 SecureString 对象完成了工作，就应当调用 Marshal 类上的静态 ZeroFreeBSTR 方法，用 0 填充并释放在从 SecureString 中提取数据时分配的任何内存。作为附加的安全措

施，你应当调用 SecureString 对象的 Clear 方法，从内存中清零加密的字符串。如果你将 SecureString 对象置为只读，那么将无法调用 Clear 方法清空数据。在这种情况下，要么必须调用 SecureString 对象上的 Dispose 方法（此处使用一个 using 语句块会更好），要么必须依赖垃圾收集器从内存中删除 SecureString 对象及其数据。

要注意，当你将一个 SecureString 对象的数据放入一个不安全的 String 中时，其数据会变得对一个恶意攻击者可见。因此，当你只是将其转换为一个不安全的 string 时，承受使用 SecureString 带来的麻烦看起来便毫无意义。但是，通过使用 SecureString，你能够降低恶意攻击者查看内存中该数据的概率。此外，某些 API 仅能接受一个 SecureString 作为参数，因此你无需其将转换为一个非安全的 String。例如，ProcessStartInfo 在其 Password 属性中接受一个作为 SecureString 对象的密码。

> SecureString 对象不是保护数据的银弹，但它是你能够为应用程序添加的另一个保护层。

11.10.4　参考

MSDN 文档中的“SecureString 类”主题。

11.11　保护流数据

11.11.1　问题

你希望使用范例 9.9（即 9.9 节）中的 TCP 服务器与范例 9.10（即 9.10 节）中的 TCP 客户端进行通信。但是，你需要加密通信并验证它在传输过程中未遭篡改。

11.11.2　解决方案

用客户端和服务器上更安全的 SslStream 类替代 NetworkStream 类。更安全的 TCP 客户端 TCPClient_SSL 的代码如例 11-10 所示。

例 11-10：TCPClient_SSL 类

```
class TCPClient_SSL
{
    private TcpClient _client = null;
    private IPAddress _address = IPAddress.Parse("127.0.0.1");
    private int _port = 5;
    private IPEndPoint _endPoint = null;

    public TCPClient_SSL(string address, string port)
    {
        _address = IPAddress.Parse(address);
        _port = Convert.ToInt32(port);
```

```
    _endPoint = new IPEndPoint(_address, _port);
}

public void ConnectToServer(string msg)
{
    try
    {
        using (client = new TcpClient())
        {
            client.Connect(_endPoint);

            using(SslStreamsslStream = newSslStream(_client.GetStream(), false,
                new RemoteCertificateValidationCallback
                        (CertificateValidationCallback)))
            {
                sslStream.AuthenticateAsClient("MyTestCert2");

                // 获得要发送消息的字节数据
                byte[] bytes = Encoding.ASCII.GetBytes(msg);
                // 发送消息
                Console.WriteLine($"Sending message to server: { msg}");
                sslStream.Write(bytes, 0, bytes.Length);

                // 获得响应
                // 用于保存响应字节数据的缓冲区
                bytes = new byte[1024];

                // 显示响应
                int bytesRead = sslStream.Read(bytes, 0, bytes.Length);
                string serverResponse = Encoding.ASCII.GetString(bytes, 0,
                      bytesRead);
                Console.WriteLine($"Server said: { serverResponse}");
            }
        }
    }
    catch (SocketException e)
    {
        Console.WriteLine
                 ($"There was an error talking to the server: {e.ToString()}");
    }
}

private bool CertificateValidationCallback(objectsender,
             X509Certificate certificate, X509Chain chain,
             SslPolicyErrors sslPolicyErrors)
{
    if (sslPolicyErrors == SslPolicyErrors.None)
    {
        return true;
    }
    else
    {
        if (sslPolicyErrors == SslPolicyErrors.RemoteCertificateChainErrors)
        {
            Console.WriteLine("The X509Chain.ChainStatus returned an array " +
```

```
                    "of X509ChainStatus objects containing error information.");
            }
            else if (sslPolicyErrors ==
                    SslPolicyErrors.RemoteCertificateNameMismatch)
            {
                Console.WriteLine(
                        "There was a mismatch of the name on a certificate.");
            }
            else if (sslPolicyErrors ==
                    SslPolicyErrors.RemoteCertificateNotAvailable)
            {
                Console.WriteLine("No certificate was available.");
            }
            else
            {
                Console.WriteLine("SSL Certificate Validation Error!");
            }
        }
        Console.WriteLine(Environment.NewLine +
                        "SSL Certificate Validation Error!");
        Console.WriteLine(sslPolicyErrors.ToString());

        return false;
    }
}
```

用于更安全的 TCP 服务器 TCPServer_SSL 的代码如例 11-11 所示。

例 11-11：TCPServer_SSL 类

```
class TCPServer_SSL
{
    private TcpListener _listener = null;
    private IPAddress _address = IPAddress.Parse("127.0.0.1");
    private int _port = 55555;

    #region CTORs
    public TCPServer_SSL()
    {
    }

    public TCPServer_SSL (string address, string port)
    {
        _port = Convert.ToInt32(port);
        _address = IPAddress.Parse(address);
    }
    #endregion // CTORS

    #region Properties
    public IPAddress Address
    {
        get { return _address; }
        set { _address = value; }
    }

    public int Port
```

```
{
    get { return _port; }
    set { _port = value; }
}
#endregion

public void Listen()
{
    try
    {
       _using(_listener = new TcpListener(_address, _port))
        {
            // 启动服务器
            listener.Start();

            // 进入监听循环
            while (true)
            {
                Console.Write("Looking for someone to talk to... ");

                // 等待连接
                TcpClient newClient = _listener.AcceptTcpClient();
                Console.WriteLine("Connected to new client");

                // 创建线程以处理客户端请求
                ThreadPool.QueueUserWorkItem(new WaitCallback(ProcessClient),
                                         newClient);
            }
        }
    }
    catch (SocketException e)
    {
        Console.WriteLine($"SocketException: {e}");
    }
    finally
    {
        // 关闭服务器
        _listener.Stop();
    }

    Console.WriteLine("Hit any key (where is ANYKEY?) to continue...");
    Console.Read();
}

private void ProcessClient(object client)
{
    using (TcpClient newClient = (TcpClient)client)
    {
        // 用于读取数据的缓冲区
        byte[] bytes = new byte[1024];
        string clientData = null;

        using (Ssl Stream sslStream = new SslStream(newClient.GetStream()))
        {
             sslStream.AuthenticateAsServer(GetServerCert("MyTestCert2"), false,
```

```
                                         SslProtocols.Default, true);
            // 循环以接收客户端发送的所有数据
            int bytesRead = 0;
            while ((bytesRead = sslStream.Read(bytes, 0, bytes.Length)) != 0)
            {
                // 将字节数据转换成ASCII字符串
                clientData = Encoding.ASCII.GetString(bytes, 0, bytesRead);
                Console.WriteLine($"Client says: {clientData}");
                // 感谢他们的输入
                bytes = Encoding.ASCII.GetBytes("Thanks call again!");

                // 发送回响应
                ssl Stream.Write(bytes, 0, bytes.Length);
            }
        }
    }
}

private static X509Certificate GetServerCert(string subjectName)
{
    X509Store store = new X509Store(StoreName.My, StoreLocation.LocalMachine);
    store.Open(OpenFlags.ReadOnly);
    X509CertificateCollection certificate =
            store.Certificates.Find(X509FindType.FindBySubjectName,
                                    subjectName, true);
    if (certificate.Count > 0)
        return (certificate[0]);
    else
        return (null);
}
}
```

11.11.3 讨论

关于 TCP 服务器和客户端的内部工作以及如何运行这些应用程序的更多信息，请参见范例 9.9（即 9.9 节）和范例 9.10（即 9.10 节）。在本范例中，我们只阐述将 TCP 服务器和客户端转换为使用 SslStream 对象进行安全通信所需的修改。

SslStream 对象使用 SSL 协议提供了一种发送数据的安全加密信道。不过，加密只是 SslStream 对象中的内置安全特性之一。SslStream 的另一个特性是它检测对数据的恶意甚至意外修改。尽管数据被加密了，它在传输过程中还是有可能被修改。为了确定这种情况是否发生过，数据在发送前要使用一个散列进行签名。当接收到它时，对数据重新计算散列并比较两个散列值。如果两个散列值相等，那么到达的信息是完好无损的；如果散列值不等，那么表明数据在传输过程中已被修改了。

SslStream 对象还具有使用客户端和 / 或服务器证书的能力，以验证客户端和 / 或服务器，而且如果客户端还需要向服务器证明身份，那么它允许客户端向服务器传递一个证书。这些证书被用于证明发行者的身份。例如，如果一个客户端使用 SSL 与服务器连接，那么服务器必须向客户端提供一个证书，证明此服务器就是它所说的服务器。证书必须由可信的权威方发出。所有可信的证书都存储在客户端的根证书存储中。

为了确保 TCP 服务器和客户端能够成功地进行通信，你需要建立一个 X.509 证书，用于验证 TCP 服务器。为此，使用 makecert.exe 程序建立一个测试证书。此实用程序随 Visual Studio 一起安装，并且必须从 Admin Visual Studio 命令提示符下运行。创建一个简单证书的语法，如下所示。

```
makecert -r -pe -n "CN=CSharpCookBook.net" -a sha512 -len 4096
        -cy authority -sv CSCBNet.pvk CSCBNet.cer
```

命令选项的定义如下所示。

- `-r`

  证书将会是自我签名的。自我签署证书通常由网站的开发者创建并签名，在其网站被产品化之前便于其网站的测试。自我签名证书不能提供证明站点合法的证据。

- `-pe`

  证书的密钥是可导出的，以便将其包括在证书之内。

- `-n "CN=MyTestCert2"`

  发布者的证书名称。名称跟在 `"CN="` 文本后面。

- `-a sha512`

  用于创建数字签名的算法。`sha512` 是可用算法中最强的。

- `-len 4096`

  密钥长度的位数。

- `-cy authority`

  此证书的类型。类型可以是 `end`（最终实体）或 `authority`（证书颁发机构）。

- `-sv CSCBNet.pvk`

  将要为使用者生成的密钥文件的名称。

makecert.exe 程序最后的参数是输出文件名，此例中为 CSCBNet.cer。这将在硬盘上当前工作目录的此文件中生成证书。此外，第二个生成的文件为 CSCBNet.pvk。这是密钥文件。密钥文件和证书文件需要转换为个人信息交换（.pfx）文件。这可以通过在 Admin Visual Studio 命令提示符下运行 Pvk2Pfx.exe 工具来实现，如下所示。

```
pvk2pfx.exe -pvk CSCBNet.pvk -spc CSCBNet.cer -pfx CSCBNet.pfx -po CSCB
```

选项的定义如下所示。

- `-pvk`

  密钥文件名。

- `-spc`

  证书文件名。

- `-pfx`

  生成的个人信息交换文件名。

- `-po`

  生成的个人信息交换文件的新密码。

下一个步骤涉及打开 Windows 资源管理器并右键单击 CSCBNet.cer 文件。这会显示一个弹出菜单。单击 Install Certificate 菜单项，将会启动一个向导，允许你将该 .cer 文件导入到证书存储中。向导的第一个对话框如图 11-1 所示。单击“下一步”按钮。



图 11-1：证书导入向导的第一步

向导的下一步允许你选择希望安装自己证书的证书存储。这个对话框如图 11-2 所示。保持默认状态并单击“下一步”按钮。

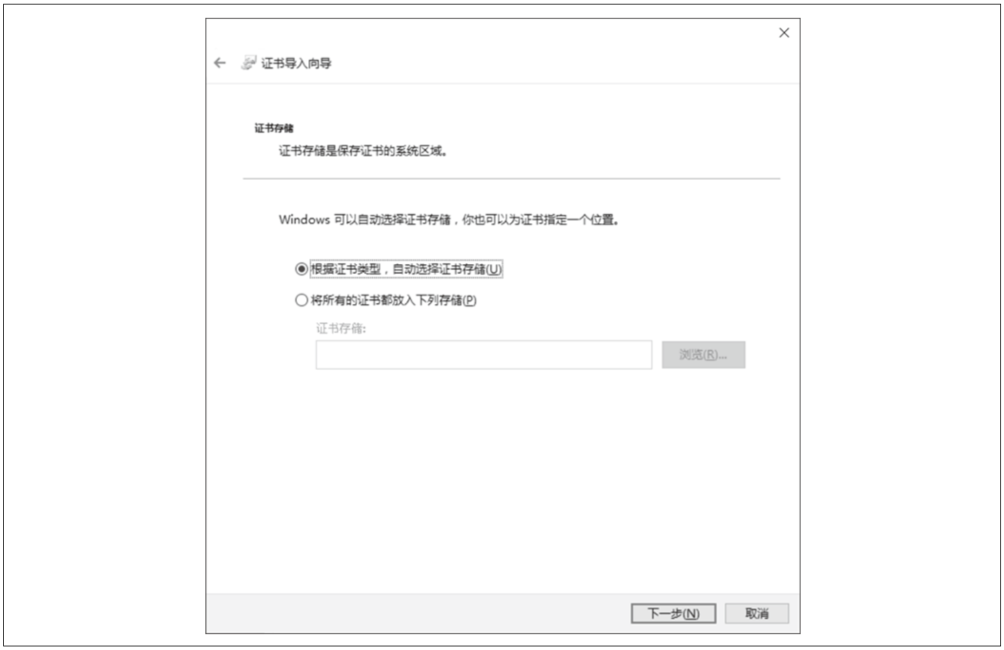


图 11-2：在证书导入向导中指定证书存储区

向导的最后一步如图 11-3 所示。在这个对话框中，单击“完成”按钮。



图 11-3：证书导入向导的最后一步

单击“完成”按钮后，将会看到如图 11-4 所示的消息对话框，表明导入成功。



图 11-4：证书导入成功消息

一旦成功导入了证书文件，你需要使用证书导入向导将 .pfx 文件导入。右键单击 CSCBNet.pfx 文件，将显示一个弹出式菜单。单击 Install PFX 菜单项，将会启动向导。向导中的第一个对话框如图 11-5 所示。保持默认设置并单击“下一步”。



图 11-5：在证书导入向导中指定密钥的存储区

此向导的下一步如图 11-6 所示，要求你选择一个 .pfx 文件导入。使用浏览按钮浏览文件，然后单击“下一步”。

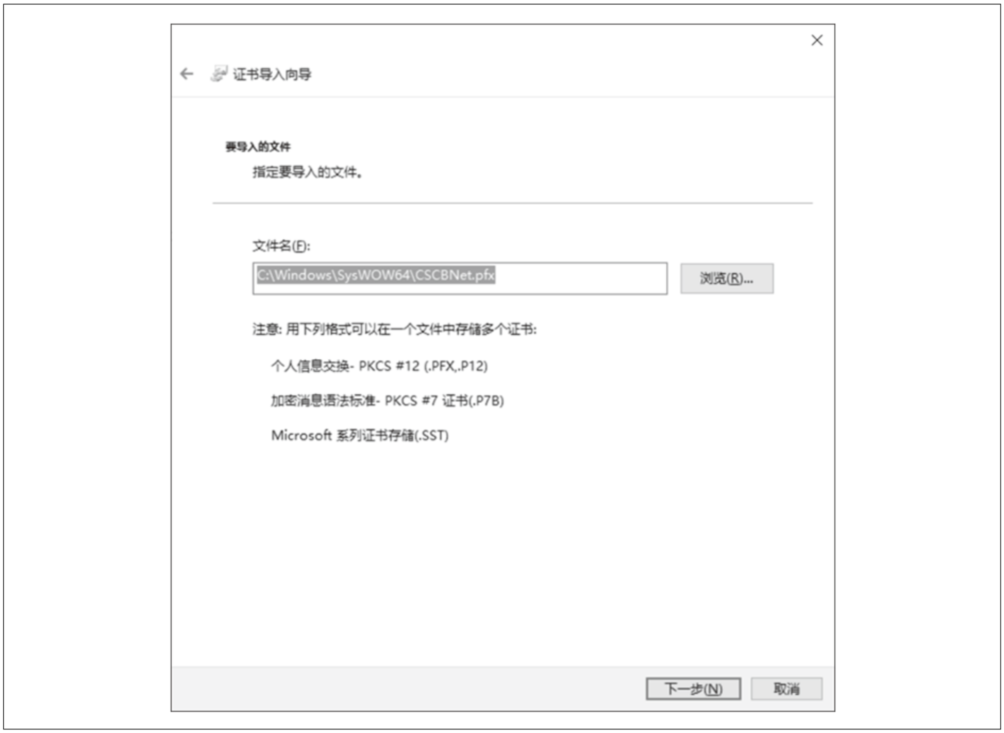


图 11-6：指定将要导入的证书存储区的个人信息交换文件

下一步如图 11-7 所示，要求提供创建此 .pfx 文件时使用的密码。注意，此密码是我们在 Pvk2Pfx.exe 命令行工具中使用的那个。实际的密码通过 `-po` 选项开关传入到此工具。对于我们的示例来说，我们使用 `CSSB` 作为密码。在此向导页上的文本框中键入密码并单击“下一步”。



图 11-7：输入个人信息交换文件的密码

接下来的一步如图 11-8 所示，要求你选择要在其中存储此 .pfx 信息的证书存储区。保持默认设置并单击“下一步”。

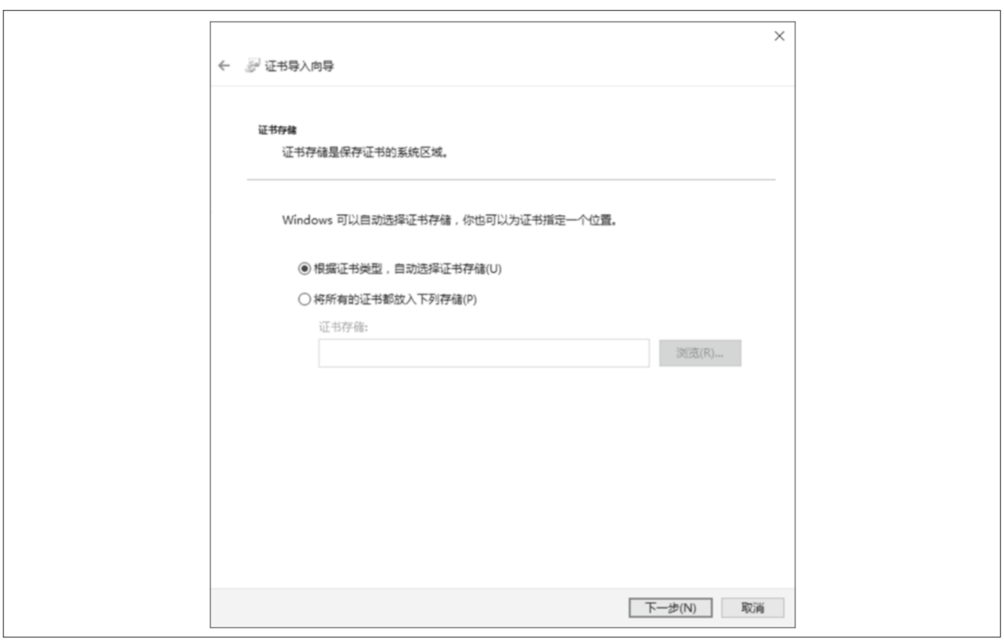


图 11-8：在证书导入向导中指定个人信息交换文件的证书存储区

向导的最后一步如图 11-9 所示，只是显示了在向导前述页面中指定的信息。单击“完成”按钮完成导入。在单击“完成”按钮后，将会看到图 11-4 中的消息对话框，表明导入成功。



图 11-9：个人信息交换文件导入成功的消息

此时，你可以运行 TCP 服务器和客户端，它们应当能够成功地通信。

要在 TCP 服务器项目中使用 `SslStream`，需要创建一个新 `SslStream` 对象以封装 `TcpClient` 对象。

```
SslStream SslStream = new SslStream(newClient.GetStream());
```

在你可以使用这个新的流对象之前，必须使用下列代码验证服务器。

```
SslStream.AuthenticateAsServer(GetServerCert("MyTestCert2"),
                               false, SslProtocols.Default, true);
```

`GetServerCert` 方法找出用于验证服务器的服务器证书。要注意传递给该方法的名称；它与 makecert.exe 工具使用的发布者的证书名选项开关相同（参考 `-n` 选项开关）。该证书作为一个 `X509Certificate` 对象从 `GetServerCert` 方法中返回。传递给 `AuthenticateAsServer` 方法的下一个参数是 `false`，表示不需要客户端证书。`SslProtocols.Default` 参数表示根据客户端和服务器可用的方法选择验证机制（SSL 2.0、SSL 3.0、TLS 1.0 或 PCT 1.0）。最后的参数表示证书将被检查，查看它是否已被撤消。

要在 TCP 客户端项目中使用 SslStream，可以创建一个新的 SslStream 对象。这与在 TCP 服务器项目中创建的方式有些许不同。

```
SslStream SslStream = new SslStream(_client.GetStream(), false,
        new RemoteCertificateValidationCallback(CertificateValidationCallback));
```

这个构造函数接受来自 _client 字段的一个流对象、一个 false（指示与 _client 字段相关联的流对象在调用了 SslStream 对象的 Close 方法后将被关闭）和一个验证服务器证书的委托。CertificateValidationCallback 方法在服务器证书需要验证时被调用。检查服务器证书，任何发现的错误都传递给这个委托方法，以允许你按自己的意愿进行处理。

接下来调用 AuthenticateAsClient 方法来验证服务器。

```
SslStream.AuthenticateAsClient("MyTestCert2");
```

如你所见，多做一点额外的工作，就能够用 SslStream 替代你正在使用的当前流类型，以获得 SSL 协议的益处。

11.11.4　参考

MSDN 文档中的“SslStream 类”主题。

11.12　加密web.config信息

11.12.1　问题

你需要以编程方式加密 web.config 文件中的数据。

11.12.2　解决方案

要加密 web.config 文件节（section）中的数据，可使用下列方法。

```
public static void EncryptWebConfigData(string appPath,
                                        string protectedSection,
                                        string dataProtectionProvider)
{
    System.Configuration.Configuration webConfig =
                WebConfigurationManager.OpenWebConfiguration(appPath);
    ConfigurationSection webConfigSection =
                webConfig.GetSection(protectedSection);

    if (!webConfigSection.SectionInformation.IsProtected)
    {
        webConfigSection.SectionInformation.ProtectSection(
                                                 dataProtectionProvider);
        webConfig.Save();
    }
}
```

要解密 web.config 文件节中的数据，可使用下列方法。

```
public static void DecryptWebConfigData(string appPath, string protectedSection)
{
    System.Configuration.Configuration webConfig =
                WebConfigurationManager.OpenWebConfiguration(appPath);
    ConfigurationSection webConfigSection =
                 webConfig.GetSection(protectedSection);

    if (webConfigSection.Section Information.IsProtected)
    {
        webConfigSection.SectionInformation.UnprotectSection();
        webConfig.Save();
    }
}
```

你需要在编译该代码之前将 `System.Web` 和 `System.Configuration` DLL 添加到自己的项目中。

11.12.3 讨论

要加密数据，可以使用下列参数调用 `EncryptWebConfigData` 方法。

```
EncryptWebConfigData("/WebApplication1", "appSettings",
                     "DataProtectionConfigurationProvider");
```

第一个参数是 Web 应用程序的虚拟路径，第二个参数是想要加密的节，最后一个参数是希望用于加密数据的数据保护提供者。

`EncryptWebConfigData` 方法使用传递给它的虚拟路径打开 web.config 文件。这是通过使用 `WebConfigurationManager` 类的 `OpenWebConfiguration` 静态方法来完成的。

```
System.Configuration.Configuration webConfig =
    WebConfigurationManager.OpenWebConfiguration(appPath);
```

该方法返回一个 `System.Configuration.Configuration` 对象，你使用它来获得 web.config 文件中希望加密的节。这可以通过调用 `GetSection` 方法来完成。

```
ConfigurationSection webConfigSection = webConfig.GetSection(protectedSection);
```

该方法返回一个用于加密文件节的 `ConfigurationSection` 对象。加密可以通过调用 `ProtectSection` 方法来完成。

```
webConfigSection.SectionInformation.ProtectSection(dataProtectionProvider);
```

`dataProtectionProvider` 参数是一个字符串，指示你希望使用哪个数据保护提供者加密文件节信息。两个可用的提供者是 `DpapiProtectedConfigurationProvider` 和 `RsaProtectedConfigurationProvider`。`DpapiProtectedConfigurationProvider` 类利用数据保护 API（DPAPI）加密和解密数据。`RsaProtectedConfigurationProvider` 类利用 .NET Framework 中的 `RsaCryptoServiceProvider` 类加密和解密数据。

加密文件节信息的最后一步是调用 `System.Configuration.Configuration` 对象的 `Save` 方法。这可保存对 web.config 文件的修改。如果未调用该方法，那么加密的数据将不会被保存。

要解密 web.config 文件中的数据，可以调用带有下列参数的 DecryptWebConfigData 方法。

```
DecryptWebConfigData("/WebApplication1", "appSettings");
```

第一个参数是 Web 应用程序的虚拟路径，第二个参数是你希望解密的文件节。

DecryptWebConfigData 方法的运行方式与 EncryptWebConfigData 方法非常相似，不同点是前者会调用 UnprotectSection 方法解密 web.config 文件中的加密数据。

```
webConfigSection.SectionInformation.UnprotectSection();
```

如果你使用这项技术加密 web.config 文件中的数据，那么当 Web 应用程序访问 web.config 文件中的加密数据时，数据将被自动解密。

11.12.4　参考

MSDN 文档中的“System.ConfigurationSection.Configuration 类”主题。

11.13　获得一个更安全的文件句柄

11.13.1　问题

你希望在操作一个非托管文件句柄时拥有的安全性比简单 IntPtr 所能提供的安全性更多。

11.13.2　解决方案

使用 Microsoft.Win32.SafeHandles.SafeFileHandle 对象封装现有的非托管文件句柄。

```
public static void WriteToFileHandle(IntPtr hFile)
{
    // 将文件句柄封装在安全句柄包装对象中
    using (Microsoft.Win32.SafeHandles.SafeFileHandle safeHFile =
        new Microsoft.Win32.SafeHandles.SafeFileHandle(hFile, true))
    {
        // 使用传入的安全句柄打开一个FileStream对像
        using (FileStream fileStream = new FileStream(safeHFile,
               FileAccess.ReadWrite))
        {
           // 在开始写入前刷新以清理任何进行中的非托管操作
           fileStream.Flush();

           // 此时开始操作文件
           string line = "Using a safe file handle object";

           // 写入文件中
           byte[] bytes = Encoding.ASCII.GetBytes(line);
           fileStream.Write(bytes,0,bytes.Length);
        }
   }
   // 注意此时hFile句柄已无效
}
```

SafeFileHandle 的构造函数接受两个参数。第一个是 IntPtr，它包含指向非托管资源的句柄。第二个参数是一个布尔值，其中 true 指示句柄总是在析构期间释放，false 指示关闭强制句柄在析构期间释放的安全措施。除非你有充足的理由关闭这些安全措施，否则建议你总是将该布尔值设置为 true。

11.13.3 讨论

SafeFileHandle 对象包含指向非托管文件资源的单个句柄。与使用 IntPtr 存储一个句柄相比，该类有两个主要好处：关键终结和防止句柄重用攻击。因为 SafeFileHandle 的基类之一是 CriticalFinalizerObject，所以 SafeFileHandle 被垃圾收集器视为一个关键终结器。垃圾收集器将终结器分为两类：关键的和非关键的。非关键终结器首先运行，然后运行关键终结器。如果一个 FileStream 的终结器刷新了任何数据，那么可以认为 SafeFileHandle 对象仍是合法的，因为 SafeFileHandle 终结器确保在 FileStream 之后运行。

> FileStream 对象上的 Close 方法也将关闭其底层的 SafeFileHandle 对象。

SafeFileHandle 被归入关键终结器，因此它意味着非托管的底层句柄总会被释放（即总是会调用 SafeFileHandle.ReleaseHandle 方法），即使在 AppDomain 被损坏、关闭或线程被中止的情况下也会如此。这将防止资源句柄泄露。

SafeFileHandle 对象也有助于防止句柄重用攻击。操作系统试图积极地重复使用句柄，因此关闭一个句柄随后很快打开另一个新句柄有可能得到相同的值。攻击者利用这一点的一种方法是，强制一个线程上的一个可访问句柄关闭，而它仍可能被用在另一个线程上，这是希望句柄很快能被重新利用并用作一个指向新资源的句柄，很可能是一个攻击者没有权限去访问的句柄。如果应用程序仍然拥有原始句柄并主动地作用它，那么数据损坏就可能会成为一个问题。

由于该类继承自 SafeHandleZeroOrMinusOneIsInvalid 类，0 或 -1 的句柄值被认为是无效的句柄。

11.13.4 参考

MSDN 文档中的“Microsoft.Win32.SafeHandles.SafeFileHandle 类”主题。

11.14 保存密码

11.14.1 问题

你需要为你的应用程序的用户以安全和可靠的方式保存密码。然而，你不希望任何拥有提升权限的人，例如系统管理员，能够有办法来解密存储的密码。此外，如果此信息被攻击者盗取，你希望他解开原始密码的难度尽可能大一些。

11.14.2　解决方案

与其使用双向加密算法来加密密码，这类算法使用正确的密钥就能够解密密码，我们将使用带有 salt 值（盐值）的单向散列算法以一种更安全的方式来存储密码。我们将比较散列值，而不是比较明文密码，从而隐藏真正的密码不被窥视。

> 本范例使用范例 11.10（即 11.10 节）中的方法，最明显的是 `CreateSecureString` 和 `ReadSecureString` 方法。

我们以创建一个接受明文密码的方法开始，返回唯一的 salt 值（作为 `out` 参数）和经过散列算法处理并随机生成的密码（作为返回值）。

```
const int HASH_ITERATIONS = 43;
const string HASH_ALGORITHM = "SHA-512";
const int SALT_LENGTH = 64;

public static SecureString GeneratePasswordHashAndSalt(SecureString passwd,
                                                       out SecureString salt)
{
    // 首先生成我们将用来散列的唯一salt
    salt = GenerateSalt();

    // 创建加盐的散列
    string hashedPwd = GenerateHash(passwd, salt);

    return CreateSecureString(hashedPwd);
}
```

接下来我们将编写生成一个在加密方面的强随机数字的方法，该随机数字将用作 salt 值。

```
private static SecureString GenerateSalt()
{
    RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();

    byte[] salt = new byte[SALT_LENGTH];
    rng.GetBytes(salt);

    return CreateSecureString(Convert.ToBase64String(salt));
}
```

当然，我们需要一个接受未经散列处理的密码和上述 `GenerateSalt` 方法生成的 salt 值的方法，然后返回最终的密码 /salt 值组合的散列。

```
private static string GenerateHash(SecureString clearTextData, SecureString salt)
{
    if (salt?.Length > 0)
    {
        // 在散列前组合密码和salt值
        byte[] clearTextDataArray =
                  Encoding.UTF8.GetBytes(ReadSecureString(clearTextData));
```

```
            byte[] clearTextSaltArray =
                Convert.FromBase64String(ReadSecureString(salt));

            byte[] clearTextDataSaltArray = new byte[clearTextDataArray.Length +
                                                     clearTextSaltArray.Length];
            Array.Copy(clearTextDataArray, 0, clearTextDataSaltArray,
                       0, clearTextDataArray.Length);
            Array.Copy(clearTextSaltArray, 0, clearTextDataSaltArray,
                       clearTextDataArray.Length, clearTextSaltArray.Length);

            // 使用一个安全的散列算法
            HashAlgorithm alg = HashAlgorithm.Create(HASH_ALGORITHM);

            byte[] hashedPwd = null;

            for (int index = 0; index < HASH_ITERATIONS; index++)
            {
                if (hashedPwd == null)
                {
                    // 明文密码的初始散列
                    hashedPwd = alg.ComputeHash(clearTextDataSaltArray);
                }
                else
                {
                    // 为增加的熵重新计算散列的散列
                    hashedPwd = alg.ComputeHash(hashedPwd);
                }
            }

            return Convert.ToBase64String(hashedPwd);
        }
        else
        {
            throw new ArgumentException(
               $"Salt parameter {nameof(salt)} cannot be empty or null. " +
                "This is a security violation.");
        }
    }
```

此 GenerateHash 方法只是将密码和 salt 值组合成单一的 byte[]，然后计算此组合值的散列。为了额外的安全性，散列值重复多次计算散列。散列迭代次数由 HASH_ITERATIONS 常量控制。

一旦创建了最终的散列加盐的密码值，我们就需要在数据存储中为此用户保存此值以及唯一的 salt 值。下列伪代码为你展示了大体思路。你可以修改此代码以用于任何数据存储。

```
    public static void SaveHashedPassword(string userName, SecureString pwdHash,
                                          SecureString salt)
    {
        string base64PwdHash = ReadSecureString(pwdHash);
        string base64Salt = ReadSecureString(salt);
        // 保存到DB
        // INSERT users ('user', 'pwd', 'salt', ...)
        //              (userName, base64PwdHash, base64Salt, ...)}
```

pwdHash 和 salt 参数应分别从 GeneratePasswordHashAndSalt 方法的返回值和 out 参数得来。

现在，我们可以创建散列加盐的密码了，我们需要一种方法来比较用户在他的应用程序登录窗体的密码文本框中输入的密码和存储在数据存储中的散列值。下面的方法将用户输入的密码散列加盐，然后将此值与存储在数据存储中同一用户的值（即用户创建的原始散列加盐的密码）进行比较。

```
public static bool ComparePasswords(SecureString storedHashedPwd,
                                    SecureString storedSalt,
                                    SecureString clearTextPwd)
{
    try
    {
        // 首先使用同样的技术来散列明文密码
        byte[] userEnteredHashedPwd =
                 Convert.FromBase64String(GenerateHash(clearTextPwd,
                     storedSalt));

        // 获得保存的散列加盐的密码
        byte[]originalHashedPwd =
                 Convert.FromBase64String(ReadSecureString(storedHashedPwd));

        // 现在比较两个散列值
        // 如果为true,说明用户输入的密码是正确的
        if (userEnteredHashedPwd.SequenceEqual(originalHashedPwd))
            return true;
    }
    catch(ArgumentException ae)
    {
        // 此处你应该记录这一错误并返回false
        Console.WriteLine(ae.Message);
        return false;
    }

    return false;
}
```

调用此方法时，必须从最初存储到的数据存储中获取 storedHashedPwd 和 salt 参数。起初，我们使用 SaveHashedPassword 方法中的伪代码保存了这些值。下面是读取这些值的另一个伪代码方法。

```
public static void RetrieveHashedPasswordAndSalt(string userName,
                                                 out SecureString
                                                     storedHashedPwd,
                                                 out SecureString storedSalt)
{
    // 从DB中读取
    // SELECT pwd, salt FROM users WHERE user = ?
    // SetString(userName);

    storedHashedPwd = CreateSecureString(getFromResultSet("pwd"));
    storedSalt = CreateSecureString(getFromResultSet ("salt"));
}
```

和上面一样，你应该修改这些伪代码来处理特定的数据存储。

11.14.3 讨论

在进入如何使用这些代码的细节之前，让我们讨论一下在此代码中使用的常量值。

```
const int HASH_ITERATIONS = 43;
const string HASH_ALGORITHM = "SHA-512";
const int SALT_LENGTH = 64;
```

首先，HASH_ITERATIONS 值简单地定义了明文密码 /salt 组合将进行多少次散列运算。在此例中，密码 /salt 值进行散列处理，生成的散列再次进行散列；如此反复，总共 43 次。如果你需要在散列中有更多熵，则应该增加此值；它很容易能增加至 100、200、500 甚至 1000。但是，要记住这需要处理能力来创建这些散列值，攻击者（可能使用僵尸网络）可以强行导致许多散列生成，从而导致你的应用程序拒绝服务。

> 当提示用户注册或登录时显示一个验证码，并在几次登录失败后锁定用户。这个办法可以防止或威慑专注于使你的服务器忙于生成散列值的拒绝服务攻击。

HASH_ALGORITHM 值定义了要使用的散列算法。使用 SHA-256 或 SHA-512 是安全的，不过使用 SHA-512 更加安全。请不要使用容易破解的散列算法，例如 MD5 或 SHA-1，因为它们将显著减少攻击者破解你的散列所用的时间。事实上，不要使用任何强度低于 SHA-256 的算法。

最后，SALT_LENGTH 值是将要组成 salt 值的字节数。这些字节由一个密码强随机数生成器生成。此处选择的 salt 长度为 64 字节，但也可以更长或更短。我们选择 64，因为这是散列过的密码和 salt 值相同的长度，迫使攻击者在对散列使用彩虹表或反向查找表前确定哪一个是 salt 值，哪一个是散列值。如果你决定使用 SHA-256，那么可以将 SALT_LENGTH 减小到 32 字节以与散列值的大小相等。

继续在你的应用中实现此代码，有两个地方应当使用此代码：站点的用户注册和登录表单中。首先，我们来逐步讨论注册过程。

(1) 用户选择注册此网站的用户名和密码。
(2) 此网站要求用户输入用户名和密码，然后传递到 GeneratePasswordHashAndSalt 方法，为此用户生成唯一的 salt 值和散列加盐的密码。
(3) 将用户输入的用户名与数据存储进行验证，以确定是否存在与现有用户同名的用户。
(4) 如果此前没有同名用户存在，在数据存储中存储用户名、散列加盐的密码和唯一的 salt 值。

> 对于本范例，我们假设在注册和登录窗体上使用 System.Windows.Controls.PasswordBox 控件。此控件可以在 PresentationFramework.dll 中找到。此控件具有一个内置的属性 SecurePassword，使我们能够获取存储在 SecureString 对象中而不是一个普通字符串对象中的密码。

代码将如下所示。

```
public bool Register()
{
    try
    {
        ...

        SecureString salt;
        SecureString pwdHash =
          GeneratePasswordHashAndSalt(myRegPasswordTextBox.SecurePassword,
              out salt);

        // 测试以确保此用户名可用于注册
        if (UserDoesNotExist(myRegUserNameTextBox.Text))
        {
            SaveHashedPassword(userName, pwdHash, salt);
            return true;
        }
        else
        {
           return false;
        }
    }
    catch(Exception e)
    {
        // 发生了错误,登录失败
        return false;
    }
}
```

我们调用的第一个方法是 `GeneratePasswordHashAndSalt`，一是为了为此用户生成新的唯一 salt 值，二是为了对用户注册所用的密码进行加盐和散列。

> 重要的是，为每个用户生成唯一的 salt 值。为每个用户使用相同的 salt 是不安全的，因为它使攻击者更加容易破解所有散列的密码。攻击者要做的就是确定一个 salt 值并将其应用于其生成的每个散列值。

在此方法中，我们做的最后一件事是测试，以确保在数据存储中此用户名并不存在。如果它不存在，我们继续使用 `SaveHashedPassword` 方法在数据存储中存储此用户名、散列加盐的密码和唯一 salt。否则注册过程将中断，用户必须输入一个不同的用户名。

以下是用户返回到该网站，并试图用凭据登录的过程。

(1) 用户输入自己的用户名和密码。
(2) 从数据存储中获得此用户唯一的 salt 值和最初散列加盐的密码。
(3) 用户输入网站的密码（在步骤 1 中得到）和此用户的唯一 salt 值（在步骤 2 中得到）以及散列加盐的密码（同样在步骤 2 中得到）传入 `ComparePasswords` 方法。
(4) `ComparePasswords` 方法简单地使用为此特定用户存储的唯一 salt 值加盐并散列用户密码，然后将返回的散列与此用户存储的初始散列相比较。

(5) 如果散列值完全相同，用户就可以继续进行身份验证；否则，禁止该用户进行身份验证。

代码如下所示。

```
public bool Login()
{
    try
    {
        ...

        string userName = myLoginUserNameTextBox.Text;

        SecureString storedHashedPwd;
        SecureString storedSalt;
        RetrieveHashedPasswordAndSalt(userName, out storedHashedPwd,
            out storedSalt);


        if (ComparePasswords(storedHashedPwd, storedSalt,
                             myLoginPwdTextBox.SecurePassword))
        {
            // 密码散列符合
            return true;
        }
        else
        {
            // 密码散列不符,登录失败
            return false;
        }
    }
    catch(Exception e)
    {
        // 发生了错误,登录失败
        return false;
    }
}
```

首先，这段代码使用由用户输入的用户名，通过 `RetrieveHashedPasswordAndSalt` 方法从数据存储中获得散列加盐的密码以及用户的唯一 salt 值。将这两个值与用户在登录窗体中输入的密码一起传到 `ComparePasswords` 方法。此方法使用 `RetrieveHashedPasswordAndSalt` 方法返回的相同 salt 值散列并加盐用户在登录窗体中输入的密码。如果用户在登录窗体中输入的散列加盐密码与存储在数据存储中的散列加盐密码相同，那么密码匹配并且允许身份验证过程继续进行。否则，身份验证失败。

11.14.4 参考

MSDN 文档中的“`System.Windows.Controls.PasswordBox` 类”“`System.Security.Cryptography.RNGCryptoServiceProvider` 类”和“`System.Security.SecureString` 类”主题。

第 12 章

线程、同步和并发

12.0 简介

线程（thread）代表程序中的单个执行逻辑流程。有些程序只需要一个线程即可高效执行，但许多程序需要多个线程，这就是本章将要讨论的内容。.NET 中的线程允许你构建出快速响应并且高效的应用程序。许多应用程序需要同时执行多个动作（比如用户界面交互和数据处理），而线程则提供了完成这项工作的能力。能够允许你的应用程序执行多项任务是应用程序设计中一项非常具有自由性但同时也非常复杂的问题。一旦在应用程序中采用多个线程执行，就需要开始考虑应用程序中哪些数据需要受到保护以防止多重并行访问、哪些数据可能导致线程出现互相依赖从而导致*死锁*（deadlocking，即线程 A 拥有线程 B 正在等待的资源，而线程 B 拥有线程 A 正在等待的资源），以及如何存储期望与各个独立线程相关联的数据。你还需要考虑处理线程时的*竞态条件*（race condition）。竞态条件发生在两个线程同时访问一个共享变量时。两个线程读取变量并且得到相同的值，然后竞争哪一个线程能够最后写入到共享变量中。最后一个写入到变量的线程“取胜”，因为它覆盖了第一个线程写入的值。你将会探究这些问题中的一部分，从而有助于利用 .NET Framework 的这一美妙的功能。你也将看到在设计和创建多线程软件过程中需要认真考虑的领域和需要关注的事项。

同步（synchronization）是关于协调线程或进程之间的活动，并确保被多个线程或进程访问的数据一直有效。同步允许线程和进程步调一致地操作。理解允许你在程序中执行多个线程的构造给了你创建能够更好地利用可用资源、更具扩展性的应用程序的能力。

并发（concurrency）是关于程序的各个方面的合作和串联工作，以实现目标。当操作在你的应用程序中并发运行时，在同一时间内会发生多个动作。并发由线程同步来促进。

12.1　创建每线程静态字段

12.1.1　问题

静态字段默认在一个应用程序域内的多个线程之间共享。你需要允许每个线程拥有自己的非共享静态字段副本，从而使静态字段能在每个线程上更新。

12.1.2　解决方案

使用 ThreadStaticAttribute 将任意 static 字段标记为线程间不可共享的。

```
public class Foo
{
    [ThreadStaticAttribute()]
    public static string bar = "Initialized string";
}
```

12.1.3　讨论

默认情况下，静态字段在同一应用程序域中访问这些字段的所有线程间共享。为了明白这一点，创建一个带有称为 bar 的静态字段和一个静态方法的类来访问并显示包含在该字段中的值。

```
private class ThreadStaticField
{
    public static string bar = "Initialized string";

    public static void DisplayStaticFieldValue()
    {
        string msg = $"{Thread.CurrentThread.GetHashCode()}" +
            $"{ contains static field value of: {ThreadStaticField.bar} ";
        Console.WriteLine(msg);
    }
}
```

接下来，创建一个测试方法在当前线程中和一个新创建的线程中访问该静态字段。

```
private static void TestStaticField()
{
    ThreadStaticField.DisplayStaticFieldValue();

    Thread newStaticFieldThread =
        new Thread(ThreadStaticField.DisplayStaticFieldValue);

    newStaticFieldThread.Start();

    ThreadStaticField.DisplayStaticFieldValue();
}
```

代码显示的输出大致如下所示。

```
9 contains static field value of: Initialized string
10 contains static field value of: Initialized string
9 contains static field value of: Initialized string
```

在上述例子中，当前线程的散列值为 9，而新线程的散列值为 10。这些值会随系统的不同而不同，注意到两个线程都访问同一个静态 bar 字段。接下来，向静态字段添加 ThreadStaticAttribute。

```
private class ThreadStaticField
{
    [ThreadStaticAttribute()]
    public static string bar = "Initialized string";

    public static void DisplayStaticFieldValue()
    {
        string msg = $"{Thread.CurrentThread.GetHashCode()}" +
            $"{ contains static field value of: {ThreadStaticField.bar} ";
        Console.WriteLine(msg);
    }
}
```

现在，显示的输出大致如下所示。

```
9 contains static field value of: Initialized string
10 contains static field value of:
9 contains static field value of: Initialized string
```

注意，新线程对于静态字段 bar 返回 null。这是预料之中的。bar 字段仅在访问它的第一个线程中被初始化，在所有其他线程中，这个字段被初始化为 null，因此，必须要在使用之前初始化所有线程中的 bar 字段。

> 要记住，在任何线程中使用标记有 ThreadStaticAttribute 的静态字段之前，都要进行初始化；也就是说，应当在传递给 ThreadStart 委托的方法中初始化该字段。应确保不使用先前代码中的字段初始化器来初始化静态字段，因为仅有一个线程会得到该初始值。

bar 字段在用于访问该字段的第一个线程之前被初始化为 "Initialized string" 字符串字面量。在先前的测试代码中，bar 字段首先被访问，因此它在当前线程中被初始化。假设你删除了 TestStaticField 方法的第一行，如下所示。

```
private static void TestStaticField()
{
    //ThreadStaticField.DisplayStaticFieldValue();

    Thread newStaticFieldThread =
        new Thread(ThreadStaticField.DisplayStaticFieldValue);

    newStaticFieldThread.Start();

    ThreadStaticField.DisplayStaticFieldValue();
}
```

现在代码显示的输出大致如下所示。

```
10 contains static field value of: Initialized string
9 contains static field value of:
```

当前线程并没有最先访问 bar 字段，因此也不初始化它。然而，当新线程首次访问它时，对它进行了初始化。

要注意，添加一个静态构造函数来初始化标记有该特性的静态字段仍然会遵循相同的行为。静态构造函数在每个应用程序域中仅运行一次。

12.1.4　参考

MSDN 文档中的“ThreadStaticAttribute 特性”和“static 修饰符（C#）”主题。

12.2　对类成员提供线程安全的访问

12.2.1　问题

你需要通过访问器函数给内部成员变量提供线程安全的访问。

下面的 NoSafeMemberAccess 类展示了三个方法：ReadNumericField、IncrementNumericField 和 ModifyNumericField。虽然三个方法都访问了内部 numericField 成员，但对于多线程访问而言，这些访问目前是不安全的。

```
public static class NoSafeMemberAccess
{
    private static int numericField = 1;

    public static void IncrementNumericField()
    {
        ++numericField;
    }

    public static void ModifyNumericField(int newValue)
    {
        numericField = newValue;
    }

    public static int ReadNumericField() => (numericField);
}
```

12.2.2　解决方案

NoSafeMemberAccess 可能会用于多线程应用程序，因此它必须修改成线程安全的。想想看，如果多线程同时调用 IncrementNumericField 方法会发生什么。有可能调用了两次 IncrementNumericField，而 numericField 仅更新了一次。为防止此类事情发生，你通过创建一个能够在代码临界区加锁的对象来修改这个类。

```
public static class SaferMemberAccess
{
    private static int numericField = 1;
    private static object syncObj = new object();

    public static void IncrementNumericField()
    {
        lock(syncObj)
        {
            ++numericField;
        }
    }

    public static void ModifyNumericField(int newValue)
    {
        lock (syncObj)
        {
            numericField = newValue;
        }
    }

    public static int ReadNumericField()
    {
        lock (syncObj)
        {
            return (numericField);
        }
    }
}
```

在 syncObj 对象上使用 lock 语句让你同步了对 numericField 成员的访问。这样就会使这三个方法对于多线程的访问都是安全的。

12.2.3 讨论

可以使用 lock 关键字将一块代码标记为临界区。lock 关键字不能在公共类型或者程序控制之外的实例上使用，否则会导致死锁。例如使用 this 指针、类的类型对象（typeof(MyClass)）或者一个字符串文本（"MyLock"）。如果你只是试图保护公共静态方法中的代码，那么也可使用带有 MethodImplOptions.Synchronized 值的 System.Runtime.CompilerServices.MethodImpl 特性来完成。

```
[MethodImpl (MethodImplOptions.Synchronized)]
public static void MySynchronizedMethod()
{
}
```

在 SaferMemberAccess 示例中使用诸如 syncObj 之类的对象会产生同步问题。如果你在一个能被应用程序中其他对象访问的对象或类型上加锁，那么其他对象也可能会试图锁定这一相同的对象。

死锁是两个共享相同资源的程序或线程执行实际上相互阻止访问资源，导致两者同时被阻塞并停止执行的情况。

以下是死锁的简单例子。

(1) 线程 1 访问资源 A，获得其上的一个锁。

(2) 线程 2 访问资源 B，获得其上的一个锁。

(3) 线程 1 试图获得资源 B，但正在等待线程 2 释放它。

(4) 线程 2 试图获得资源 A，但正在等待线程 1 释放它。

(5) 此时这两个线程被死锁。

这说明它是锁定自己的糟糕代码，如下所示。

```
public class DeadLock
{
    public void Method1()
    {
        lock(this)
        {
            // 执行某些操作
        }
    }
}
```

当调用 `Method1` 时，它锁定了当前 `deadLock` 对象。不幸的是，任何访问 `DeadLock` 类的对象也可能会锁定它，如下所示。

```
public class AnotherCls
{
    public void DoSomething()
    {
        DeadLock deadLock = new DeadLock();
        lock(deadLock)
        {
            Thread thread = new Thread(deadLock.Method1);
            thread.Start();
            // 此处执行一些耗时的任务
        }
     }
}
```

`DoSomething` 方法获得 `deadLock` 对象上的一个锁，然后试图在另一个线程上调用 `deadLock` 对象的 `Method1` 方法，之后执行一个耗时很长的任务。在长时间的任务执行时，在 `deadLock` 对象上的锁阻止了 `Method1` 在另一线程上被调用。只有当此长时任务结束之后，执行离开 `DoSomething` 方法的临界区时，`Method1` 方法才能够获得这一对象上的锁。如你所见，这可能会变成一个在大型应用程序中执行跟踪的棘手问题。

Jeffrey Richter 采用了一种相对简单的方法来纠正这一情况，他在 2003 年 1 月 *MSDN Magazine* 的文章“Safe Thread Synchronization”中行了详细的叙述。他的解决方案是在一个要同步的类中创建一个私有字段。只有对象本身能够获得该私有字段，外部对象或类型都不能获得它。这个解决方案也是目前 MSDN 文档中针对 `lock` 关键字的推荐实践。可如

下重新编写 DeadLock 类来解决这个问题。

```
public class DeadLock
{
    private object syncObj = new object();

    public void Method1()
    {
        lock(syncObj)
        {
            // 执行某些操作
        }
    }
}
```

现在，在 DeadLock 类中锁定了内部的 syncObj，同时 DoSomething 方法锁定了 DeadLock 类的实例。这解决了死锁条件，但 DoSomething 方法仍然不应当锁在公共类型上。因此，可如下修改 AnotherCls 类。

```
public class AnotherCls
{
    private object deadLockSyncObj = new object();

    public void DoSomething()
    {
        DeadLock deadLock = new DeadLock();
        lock(deadLockSyncObj)
        {
            Thread thread = new Thread(deadLock.Method1);
            thread.Start();
            // 此处执行一些耗时的任务
        }
    }
}
```

现在，AnotherCls 类有一个属于它自己的对象来保护在 DoSomething 中对 DeadLock 类的访问，而不是在公共类型上加锁。

要清理你的代码，应当停止锁定任何对象和类型，对于你的类型或对象私有的同步对象除外，例如修正过的 DeadLock 类中的 syncObj 对象。本范例通过在 SaferMemberAccess 类中创建一个静态 syncObj 对象来使用这个模式。IncrementNumericField、ModifyNumericField 和 ReadNumericField 方法使用该 syncObj 对 numericField 字段进行同步访问。注意，如果你在 ReadNumericField 方法读取 numericField 时不需要锁，那么可以移除这个 lock 语句块，并简单地返回 numericField 字段中包含的值。

> 在你的代码中尽量减少临界区的数量能够显著提高性能。使用你所需要的数量来保护资源访问，但不要过多。

如果你需要对临界区的加锁和解锁进行更多的控制，那么可以试着使用重载的静态 Monitor.TryEnter 方法。这些方法通过引入一个超时值从而具有更大的灵活性。lock 关键字将试图无限期地获取一个临界区上的锁，但是使用 TryEnter 方法，你可以指定一个毫秒级的超时值（作为一个整数）或作为一个 TimeSpan 结构。如果获得了锁，那么 TryEnter 方法返回 true，否则返回 false。注意，仅接受单个参数的 TryEnter 方法的重载版本在任何时候都不会阻塞。不管是否获得了锁，该方法都立即返回。

使用 Monitor 方法的更新类如例 12-1 所示。

例 12-1：使用 Monitor 方法

```
public static class MonitorMethodAccess
{
    private static int numericField = 1;
    private static object syncObj = new object();
    public static object SyncRoot => syncObj;

    public static void IncrementNumericField()
    {
        if (Monitor.TryEnter(syncObj, 250))
        {
            try
            {
                ++numericField;
            }
            finally
            {
                Monitor.Exit(syncObj);
            }
        }
    }

    public static void ModifyNumericField(int newValue)
    {
        if (Monitor.TryEnter(syncObj, 250))
        {
            try
            {
                numericField = newValue;
            }
            finally
            {
                Monitor.Exit(syncObj);
            }
        }
    }

    public static int ReadNumericField()
    {
        if (Monitor.TryEnter(syncObj, 250))
        {
            try
            {
                return (numericField);
```

```
                }
                finally
                {
                    Monitor.Exit(syncObj);
                }
            }

            return (-1);
        }
        [MethodImpl (MethodImplOptions.Synchronized)]
        public static void MySynchronizedMethod()
        {
        }
    }
```

注意，使用 TryEnter 方法时，你应当总是检查锁是否被实际获得；如果没有获得，代码应当等待并重试或者返回给调用者。

此时你可能会认为，所有这些方法都是线程安全的。单独来说，它们每一个都是线程安全的，但如果你尝试调用它们并期望两个方法间的同步访问会怎样呢？如果 ModifyNumericField 和 ReadNumericField 被 Thread 1 上的 Class 1 相继使用，而与此同时，Thread 2 上的 Class 2 正在使用这些方法，那么加锁或 Monitor 调用将无法阻止 Class 2 在 Thread 1 读它之前改变这个值。以下是演示此过程的一系列操作。

- Class 1，Thread 1

  用 10 调用 ModifyNumericField

- Class 2，Thread 2

  用 15 调用 ModifyNumericField

- Class 1，Thread 1

  调用 ReadNumericField，并获得 15 而不是 10

- Class 2，Thread 2

  调用 ReadNumericField，并获得期望中的 15

为了解决这一同步读写问题，调用方的类需要管理交互。外部类可以通过使用 Monitor 类在 MonitorMethodAccess 公开的同步对象 SyncRoot 上建立锁来达成此目的。

```
int num = 0;
if(Monitor.TryEnter(MonitorMethodAccess.SyncRoot,250))
{
    MonitorMethodAccess.ModifyNumericField(10);
    num = MonitorMethodAccess.ReadNumericField();
    Monitor.Exit(MonitorMethodAccess.SyncRoot);
}
Console.WriteLine(num);
```

当你在学习编写线程安全访问的代码时，温习一下防死锁算法（如由 Edsger Dijkstra 提出的银行家算法）以及阅读操作系统书籍会有助于你思考创建代码的方式以及它会如何反应。

12.2.4　参考

MSDN 文档中的“lock 语句”“Thread 类”和“Monitor 类”主题；*MSDN Magazine* 2003 年 1 月刊的文章“Safe Thread Synchronization”；维基百科中的“Banker's algorithm”和“Deadlock Prevention algorithms”。

12.3　避免沉默的线程终止

12.3.1　问题

如果未处理异常，那么从工作者线程引发的异常会导致这个线程被悄悄地终止。你需要确保处理了所有线程的所有异常。如果异常发生在这个新线程中，你希望处理它并获得它发生的通知。

12.3.2　解决方案

你必须使用 `try-catch`、`try-finally` 或 `try-catch-finally` 块向传递给 `ThreadStart` 委托的方法添加异常处理。完成这项工作的代码如例 12-2 所示。

例 12-2：防止沉默的线程终止

```
public class MainThread
{
    public void CreateNewThread()
    {
        // 创建新线程以执行并行工作
        Thread newWorkerThread = new Thread(Worker.DoWork);
        newWorkerThread.Start();
    }
}

public class Worker
{
    // 由ThreadStart委托调用来执行并行工作的方法
    public static void DoWork ()
    {
        try
        {
            // 此处执行线程的工作
            throw new Exception("Boom!");
        }
        catch(Exception e)
        {
            // 此处处理线程异常
            Console.WriteLine(e.ToString());
            // 不要重新引发异常
        }
        finally
        {
            // 此处执行线程清理
```

```
            }
        }
    }
```

12.3.3 讨论

如果在一个应用程序的主线程中发生了一个未经处理的异常，主线程会终止，你的整个应用程序也会终止。但是，生成的工作者线程中一个未经处理的异常只会终止这一个线程。这一情况在发生时不会产生任何可见的警告，应用程序将继续运行，仿佛没有任何事情发生一样；甚至更糟——可能会因为损坏的数据或者工作者线程不正确的执行和交互而开始出现奇怪的行为。

简单地为 Thread 类的 Start 方法封装异常处理程序将无法捕获新生成线程的异常。Start 方法的调用发生在当前线程的上下文中，而不是在新建线程上。一旦线程被加载运行，它也会立即返回，并它不会等待着线程完成。因此，新线程中引发的异常不会被捕获，因为它对其他任何线程都是不可见的。

如果从 catch 块中重新引发异常，那么该结构化的异常处理程序的 finally 块仍将执行。但是，在 finally 块结束后，异常会被再一次引发。重新引发的异常无法被处理，并且线程终止。如果 finally 块后存在代码，那么它将不会被执行，因为一个未经处理的异常发生了。

> 绝不要在线程内部异常处理层次中的最高点重新引发异常。因为没有异常处理程序能够捕获这一重新引发的异常，所以它会被认为是未经处理的，线程将在所有的 finally 语句块执行后终止。

如果使用 ThreadPool 和 QueueUserWorkItem 会怎么样呢？该方法仍将对你有所帮助，因为添加了将在线程内部执行的处理代码。只要确保你已经建立了所有 finally 块，那么就可以得到在其他线程里发生异常的通知，并清理任何未清理的资源，如之前所示。

为了向你的 WinForms 应用程序提供最后一次机会的异常处理程序，你需要挂接两个独立的事件。第一个事件是 System.AppDomain.CurrentDomain.UnhandledException 事件，它会捕获当前 AppDomain 中工作者线程上的所有未处理的异常；它不捕获发生在 WinForms 应用程序的主 UI 线程上的异常。参见范例 5.8（即 5.8 节）可获得关于 System.AppDomain.UnhandledException 事件的更多信息。为了捕获主 UI 线程上的异常，你还需要挂接 System.Windows.Forms.Application.ThreadException，它会捕获主 UI 线程上未经处理的异常。有关 ThreadException 事件的更多信息，请参见范例 5.7（即 5.7 节）。

12.3.4 参考

MSDN 文档中的“Thread 类”和“Exception 类”主题。

12.4 在异步委托完成时获得通知

12.4.1 问题

你需要一种方法，用来从异步调用的委托处接收完成的通知。这一方法必须允许代码继续处理，而不需要在一个循环中调用 IsCompleted 或者依赖 WaitOne 方法。由于异步委托会返回一个值，你必须能够向调用方的线程传递回这个返回值。

12.4.2 解决方案

使用 BeginInvoke 方法来启动异步委托，但使用第一个参数向异步委托传递一个回调委托，如例 12-3 所示。

例 12-3：获得一个匿名委托的完成通知

```
public class AsyncAction
{
    public void CallbackAsyncDelegate()
    {
        AsyncCallback callBack = DelegateCallback;

        AsyncInvoke method1 = TestAsyncInvoke.Method1;
        Console.WriteLine(
            $"Calling BeginInvoke on Thread {Thread.CurrentThread.ManagedThreadId}");
        IAsyncResult asyncResult = method1.BeginInvoke(callBack, method1);

        // 此处不需要轮循或使用WaitOne方法,因此返回到调用方法
        return;
    }

    private static void DelegateCallback(IAsyncResult iresult)
    {
        Console.WriteLine(
            $"Getting callback on Thread {Thread.CurrentThread.ManagedThreadId}");
        AsyncResult asyncResult = (AsyncResult)iresult;
        AsyncInvoke method1 = (AsyncInvoke)asyncResult.AsyncDelegate;

        int retVal = method1.EndInvoke(asyncResult);
        Console.WriteLine($"retVal (Callback): {retVal}");
    }
}
```

该回调委托将在异步委托完成处理时在方法被调用的线程上调用 DelegateCallback 方法。如果该线程当前正在执行其他代码，回调将等待，直到线程空闲为止。线程将继续存在，因为系统知道一个回调在挂起，所以你不需要考虑回调准备好被调用时线程不存在的情况。

以下代码定义了 AsyncInvoke 委托和异步调用的静态方法 TestAsyncInvoke.Method1。

```
public delegate int AsyncInvoke();

public class TestAsyncInvoke
{
    public static int Method1()
    {
        Console.WriteLine(
            $"Invoked Method1 on Thread {Thread.CurrentThread.ManagedThreadId}");
        return (1);
    }
}
```

为了运行异步调用，可创建一个 AsyncAction 类的实例并如下调用 CallbackAsyncDelegate 方法。

```
AsyncAction aa2 = new AsyncAction();
aa2.CallbackAsyncDelegate();
```

该代码的输出如下所示。要注意，Method1 的线程 ID 是不同的。

```
Calling BeginInvoke on Thread 9
Invoked Method1 on Thread 10
Getting callback on Thread 10
retVal (Callback): 1
```

12.4.3 讨论

代替使用 IsCompleted 属性来确定何时异步委托完成了处理（或者使用 WaitOne 方法阻塞一段时间，同时异步委托继续处理），本范例使用一个回调来指示调用线程异步委托已完成处理并且其返回值（ref 参数值和 out 参数值）已可用。

以这样的方式调用委托比简单地轮询 IsCompleted 属性来确定委托何时结束处理更加灵活和高效。在一个循环中轮询该属性时，轮询方法不能返回，并且允许应用程序继续进行处理。回调也比使用 WaitOne 方法更好，因为 WaitOne 方法会阻塞调用线程并且不允许进行处理。

本范例中的 CallbackAsyncDelegate 方法利用异步委托的 BeginInvoke 方法的第一个参数传递另一个委托。它包含了一个当异步委托完成处理时将被调用的回调方法。调用 BeginInvoke 后，这个方法可以立即返回，应用程序能够继续处理，不必在轮询循环中等待或者在异步委托运行时被阻塞。

传递给 BeginInvoke 方法第一个参数的 AsyncInvoke 委托如下定义。

```
public delegate void AsyncCallback(IAsyncResult ar)
```

创建该委托后，传入的回调方法 DelegateCallback 将在异步委托完成之后被立即调用。

```
AsyncCallback callBack = new AsyncCallback(DelegateCallback);
```

DelegateCallback 不会在 BeginInvoke 同一个线程上运行，而是在来自 ThreadPool 的一个线程上运行。该回调方法接受一个 IAsyncResult 类型的参数。你可以在该方法内将此参数类型转化为一个 AsyncResult，然后使用它来获得关于已完成的异步委托的信息，比如委

托的返回值、任何 ref 参数和任何 out 参数值。如果用于调用 BeginInvoke 的委托实例仍在作用域内，你可以只向 EndInvoke 方法传递 IAsyncResult。此外，该对象能够获得传递给 BeginInvoke 方法的第二个参数的任意状态信息。该状态信息可以是任何对象类型。

DelegateCallback 方法将 IAsyncResult 参数类型转换为一个 AsyncResult 对象，并获得初始调用的异步委托。调用该异步委托的 EndInvoke 方法来处理任何返回值、ref 参数或 out 参数。如果将任何状态对象传递给 BeginInvoke 方法的第二个参数，那么可通过下列代码获得它。

```
object state = asyncResult.AsyncState;
```

12.4.4 参考

MSDN 文档中的“AsyncCallback 委托”主题。

12.5 私有化存储线程特定的数据

12.5.1 问题

你希望存储运行时发现的线程特定的数据，这个数据应当仅对运行在该线程内的代码是可访问的。

12.5.2 解决方案

在 Thread 类上使用 AllocateDataSlot、AllocateNamedDataSlot 或 GetNamedDataSlot 方法来预留一个*线程本地存储*（thread local storage，TLS）槽。使用 TLS，你可以把一个大对象存储到线程内的一个数据槽，并且在许多不同的方法中使用它，而不必将此结构作为一个参数传递。

就本例来说，一个名为 ApplicationData 的类代表了一组可能变得非常大的数据。

```
public class ApplicationData
{
    // 应用程序数据存储在此处
}
```

在使用这个结构之前，必须在 TLS 里创建一个数据槽来存储此类。首先，调用 GetNamedDataSlot 来获取 appDataSlot。由于 appDataSlot 不存在，GetNamedDataSlot 默认会创建它。下列代码创建了一个 ApplicationData 类的实例并在名为 appDataSlot 的数据槽中保存它。

```
ApplicationData appData = new ApplicationData();
Thread.SetData(Thread.GetNamedDataSlot("appDataSlot"), appData);
```

每当你需要此类时，都可以通过调用 Thread.GetData 来获取它。下面的代码从名为 appDataSlot 的数据槽中获得 appData 结构。

```
ApplicationData storedAppData = (ApplicationData)Thread.GetData(
    Thread.GetNamedDataSlot("appDataSlot"));
```

此时，storedAppData 结构可被读取或修改。在 storedAppData 上执行动作后，必须将其放回名为 appDataSlot 的数据槽中。

```
Thread.SetData(Thread.GetNamedDataSlot("appDataSlot"), storedAppData);
```

一旦应用程序完成了对这个数据的使用，那么可通过下列方法调用从内存中释放该数据槽。

```
Thread.FreeNamedDataSlot("appDataSlot");
```

例 12-4 中的 HandleClass 类展示了如何使用 TLS 存储结构。

例 12-4：使用 TLS 存储结构

```
public class HandleClass
{
    public static void Run()
    {
        // 创建结构实例并将其存储在命名数据槽中
        ApplicationData appData = new ApplicationData();
        Thread.SetData(Thread.GetNamedDataSlot("appDataSlot"), appData);

        // 调用另一个将会使用此结构的方法
        HandleClass.MethodB();

        // 完成之后,释放此数据槽
        Thread.FreeNamedDataSlot("appDataSlot");
    }

    public static void MethodB()
    {
        // 从命名数据槽中获得此实例
        ApplicationData storedAppData = (ApplicationData)Thread.GetData(
            Thread.GetNamedDataSlot("appDataSlot"));

        // 修改ApplicationData

        // 完成数据修改后,将变化保存回命名的数据槽
        Thread.SetData(Thread.GetNamedDataSlot("appDataSlot"),
            storedAppData);

        // 调用另一个将会使用此结构的方法
        HandleClass.MethodC();
    }

    public static void MethodC()
    {
        // 从命名数据槽中获得实例
        ApplicationData storedAppData =
            (ApplicationData)Thread.GetData(Thread.GetNamedDataSlot("appDataSlot"));

        // 修改数据
```

```
        // 完成数据修改后,将变化保存回命名的数据槽
        Thread.SetData(Thread.GetNamedDataSlot("appDataSlot"), storedAppData);
    }
}
```

12.5.3 讨论

线程本地存储可以方便地存储跨方法调用可用的数据，无需用户向方法传入结构，甚至不需要了解实际是在何处创建的结构。

在一个命名 TLS 数据槽中存储的数据仅对那个线程可用；其他线程无法访问另一个线程中的命名数据槽。在这个数据槽中存储的数据在线程内的任何地方均可访问。这一设定实质上令该数据成为线程全局数据。你应该意识到 TLS 数据槽是一个有限的资源，随平台的不同而变化。

要创建一个命名数据槽，可使用静态 Thread.GetNamedDataSlot 方法。该方法接受一个定义数据槽名称的单个参数 name。名称应该是唯一的，如果已存在一个同名的数据槽，该数据槽的内容将被返回，而不会创建一个新的数据槽。这个行为会默默地发生，不会引发异常或错误代码，通知你正在使用别处已创建的数据槽。为了确保你使用的是唯一的数据槽，可使用 Thread.AllocateNamedDataSlot 方法。如果一个同名的数据槽已经存在，该方法会引发一个 System.ArgumentException。否则，它的操作与 GetNamedDataSlot 方法类似。

请注意，在进程中的每个线程上都会创建这个已命名的数据槽，而不仅仅是在调用这个方法的线程上。但是，这一事实顶多只是给你带来一点不便，因为每个数据槽中的数据都只能被包含它的线程访问。此外，如果在一个单独的线程上创建一个同名数据槽，并且你用这个名称在当前线程上调用 GetNamedDataSlot，任何线程上的任何数据槽中的数据都不会遭到破坏。

GetNamedDataSlot 返回一个用于访问数据槽的 LocalDataStoreSlot 对象。注意，这个类不能通过使用 new 关键字来创建，必须通过 Thread 类上的 AllocateDataSlot 或 Allocate-NamedDataSlot 方法之一来创建。

要在这个数据槽中存储数据，可使用静态 Thread.SetData 方法，这个方法接受传递给 data 参数的对象，并将其存储在由 dataSlot 参数定义的数据槽中。

静态 Thread.GetData 方法取回存储在数据槽中的对象，这个方法取回通过 Thread.GetNamedDataSlot 方法创建的 LocalDataStoreSlot 对象。GetData 方法然后返回存储在该特定数据槽中的对象。要注意的是，返回的对象在使用之前必须强制转换成它的原始类型。

静态方法 Thread.FreeNamedDataSlot 会释放与命名数据槽相关的内存。这个方法接受数据槽的名称作为一个字符串，然后释放与该数据槽相关的内存。要记住，当使用 GetNamedDataSlot 创建数据槽时，在该进程中其他所有运行线程上，也会创建一个数据槽。使用 GetNamedDataSlot 创建数据槽真的不是一个问题；如果同名数据槽存在，那么会返回一个引用该数据槽的 LocalDataStoreSlot 对象，不会创建新数据槽，而该数据槽中的原始数据也不会损坏。

当使用 FreeNamedDataSlot 方法时，这种情况就会成为一个问题。这个方法会释放所有线

程中与传入的数据槽名关联的内存，而不仅仅是调用所处的线程。在所有线程结束使用该数据槽中的数据之前释放数据槽可能会给应用程序带来灾难性的后果。

解决这个问题的方法之一是根本不要调用 FreeNamedDataSlot 方法。当一个线程终止时，TLS 中的所有数据槽会自动释放。不调用 FreeNamedDataSlot 的副作用是这个数据槽会被一直占用，直到垃圾收集器确定创建数据槽的线程已经结束并且数据槽可被释放为止。

如果你在编译期间知道代码所需 TLS 槽的数量，可以考虑在类的一个静态字段上使用 ThreadStaticAttribute 来建立类似 TLS 的存储。

12.5.4 参考

MSDN 文档中的“Thread Local Storage: Thread Relative Static Fields and Data Slots”“ThreadStaticAttribute 特性”和“Thread 类”主题。

12.6 使用信号量允许资源的多重访问

12.6.1 问题

你拥有一个资源，希望在给定时间内只有一定数量的客户可以访问它。

12.6.2 解决方案

使用信号量来实现对资源的资源计数访问。例如，如果你有一台 Xbox One 和一套《光环5》（*Halo 5*）的副本（资源）以及一位急于舒缓压力的开发人员（用户），那么必须同步对 Xbox One 的访问。因为 Xbox One 最多有 8 个控制器，所以在给定时间内一次最多能有 8 个人一起玩游戏。游戏规则是，当一个人的角色死亡后，就得让出控制器。

为此，可以如下使用一个名为 _XboxOne 的 Semaphore 创建一个名为 Halo5Session 的类。

```
public class Halo5Session
{
    // 一个模拟有限资源池的信号量
    private static Semaphore _XboxOne;
```

为了完成这项工作，你需要调用 Halo5Session 类上的 Play 方法，如例 12-5 所示。

例 12-5：Play 方法

```
public static void Play()
{
    // 一台XboxOne最多有8个控制器端口,所以8个人可以同时游戏
    // 我们使用8作为最大值,0作为初始值,因为我们希望玩家
    // 首先排队以等待XboxOne启动并加载游戏
    //
    using (_XboxOne = new Semaphore(0, 8, "XboxOne"))
    {
        using (ManualResetEvent GameOver =
            new ManualResetEvent(false))
        {
```

```
                //
                // 13个玩家登入以等待游戏
                //
                List<XboxOnePlayer.PlayerInfo> players =
                    new List<XboxOnePlayer.PlayerInfo>() {
                        new XboxOnePlayer.PlayerInfo { Name="Igor",Dead=GameOver},
                        new XboxOnePlayer.PlayerInfo { Name="AxeMan",Dead=GameOver},
                        new XboxOnePlayer.PlayerInfo { Name="Dr. Death",
                            Dead=GameOver},
                        new XboxOnePlayer.PlayerInfo { Name="HaPpyCaMpEr",
                            Dead=GameOver},
                        new XboxOnePlayer.PlayerInfo { Name="Executioner",
                            Dead=GameOver},
                        new XboxOnePlayer.PlayerInfo { Name="FragMan",Dead=GameOver},
                        new XboxOnePlayer.PlayerInfo { Name="Beatdown",
                            Dead=GameOver},
                        new XboxOnePlayer.PlayerInfo { Name="Stoney",Dead=GameOver},
                        new XboxOnePlayer.PlayerInfo { Name="Pwned",Dead=GameOver},
                        new XboxOnePlayer.PlayerInfo { Name="Big Dawg",
                            Dead=GameOver},
                        new XboxOnePlayer.PlayerInfo { Name="Playa",Dead=GameOver},
                        new XboxOnePlayer.PlayerInfo { Name="BOOM",Dead=GameOver},
                        new XboxOnePlayer.PlayerInfo { Name="Mr. Mxylplyx",
                            Dead=GameOver}
                        };

                foreach (XboxOnePlayer.PlayerInfo player in players)
                {
                    Thread t = new Thread(XboxOnePlayer.JoinIn);

                    // 将名称保存在线程上
                    t.Name = player.Name;
                    // 启动玩家
                    t.Start(player);
                }

                // 等待XboxOne开机并加载Halo5(3秒钟)
                Console.WriteLine("XboxOne initializing...");
                Thread.Sleep(3000);
                Console.WriteLine(
                    "Halo 5 loaded & ready, allowing 8 players in now...");

                // XboxOne拥有全部信号量计数
                // 我们调用Release(8)以开放8个槽并且
                // 允许等待的玩家进入XboxOne(信号量)
                // 最多一次8个人
                //
                _XboxOne.Release(8);

                // 等待游戏结束……
                GameOver.WaitOne();
            }
        }
    }
}
```

Play 方法做的第一件事就是创建一个最多具有 8 个资源数和一个名称 _XboxOne 的新信号量。这个信号量将被所有玩家线程使用，以获得对游戏的访问。创建一个名为 GameOver 的 ManualResetEvent 以跟踪游戏何时结束。

为了模拟开发者，你为每个玩家创建一个线程，带有它自己的 XboxOnePlayer.PlayerInfo 类实例，其中包含玩家的姓名和在 PlayerInfo 的 Dead 事件中的对用于标识玩家死亡的原始 ManualResetEvent 实例 GameOver 的引用。创建线程使用了 ParameterizedThreadStart 委托，它在构造函数中接受要在新线程上执行的方法，同时允许你直接向 Thread.Start 方法的新重载版本传递数据对象。

玩家准备好开始之后，Xbox One 便开始“初始化”，然后在信号量上调用 Release，打开玩家线程可获取的 8 个槽，然后等待，直到它从玩家的 Dead 事件激发中检测到游戏结束为止。

玩家在独立的线程上初始化，然后运行 JoinIn 方法，如例 12-6 所示。首先，它们通过名称打开 Xbox One 信号量，并获取传递给线程的数据。一旦拥有了信号量，它们就调用 WaitOne 排队等候游戏。一旦最初的 8 个槽被打开或者有玩家死亡，就对 WaitOne 的调用解除阻塞，玩家可以玩一段时间游戏，直到角色死亡为止。一旦玩家角色死亡，他们便调用信号量上的 Release，表示他们的槽现在打开了。如果信号量达到最大资源数量，就设置 GameOver 事件。

例 12-6：JoinIn 方法

```
public class XboxOnePlayer
{
    public class PlayerInfo
    {
        public ManualResetEvent Dead {get; set;}
        public string Name {get; set;}
    }

    // 玩家死亡模式
    private static string[] _deaths = new string[7]{"bought the farm",
                        "choked on a rocket",
                        "shot their own foot",
                        "been captured",
                        "fallen to their death",
                        "died of lead poisoning",
                        "failed to dodge a grenade",
                        };

    /// <summary>
    /// 线程函数
    /// </summary>
    /// <param name="info">PlayerInfo数据项</param>
    public static void JoinIn(object info)
    {
        // 通过名称打开信号量,为了让我们操作它
        using (Semaphore XboxOne = Semaphore.OpenExisting("XboxOne"))
        {

            // 获得数据对象
            PlayerInfo player = (PlayerInfo)info;
```

```
            // 每个玩家通知XboxOne他们想进行游戏
            Console.WriteLine($"{player.Name} is waiting to play!");

            // 等待XboxOne(信号量)直到让玩家获得了
            // 一个控制器
            XboxOne.WaitOne();

            // XboxOne选择了此玩家
            // (或者说信号允许了访问资源……)
            Console.WriteLine($"{player.Name} has been chosen to play. " +
                $"Welcome to your doom {player.Name}. >:)");

            // 确定一个随机值,用于确定玩家持续游戏了多久
            System.Random rand = new Random(500);
            int timeTillDeath = rand.Next(100, 1000);

            // 模拟玩家忙于玩游戏,直到角色死亡为止
            Thread.Sleep(timeTillDeath);

            // 确定玩家角色是如何死亡的
            rand = new Random();
            int deathIndex = rand.Next(6);

            // 通知此玩家已结束游戏并跳过
            Console.WriteLine($"{player.Name} has {_deaths[deathIndex]} " +
            "and gives way to another player");

            // 如果所有的端口都打开了,所有人都已玩过了游戏,游戏结束
            int semaphoreCount = XboxOne.Release();
            if (semaphoreCount == 3)
            {
                Console.WriteLine("Thank you for playing, the game has ended.");
                // 设置玩家的Dead事件
                player.Dead.Set();
            }
        }
    }
}
```

当运行 Play 方法时，将产生类似以下的输出。

```
Igor is waiting to play!
AxeMan is waiting to play!
Dr. Death is waiting to play!
HaPpyCaMpEr is waiting to play!
Executioner is waiting to play!
FragMan is waiting to play!
Beatdown is waiting to play!
Stoney is waiting to play!
Pwned is waiting to play!
Big Dawg is waiting to play!
Playa is waiting to play!
XboxOne initializing...
BOOM is waiting to play!
```

```
Mr. Mxylplyx is waiting to play!
Halo 5 loaded & ready, allowing 8 players in now...
Stoney has been chosen to play. Welcome to your doom Stoney. >:)
Executioner has been chosen to play. Welcome to your doom Executioner. >:)
Beatdown has been chosen to play. Welcome to your doom Beatdown. >:)
Pwned has been chosen to play. Welcome to your doom Pwned. >:)
Playa has been chosen to play. Welcome to your doom Playa. >:)
HaPpyCaMpEr has been chosen to play. Welcome to your doom HaPpyCaMpEr. >:)
Big Dawg has been chosen to play. Welcome to your doom Big Dawg. >:)
FragMan has been chosen to play. Welcome to your doom FragMan. >:)
Playa has been captured and gives way to another player
Stoney has been captured and gives way to another player
Pwned has been captured and gives way to another player
Big Dawg has been captured and gives way to another player
Mr. Mxylplyx has been chosen to play. Welcome to your doom Mr. Mxylplyx. >:)
BOOM has been chosen to play. Welcome to your doom BOOM. >:)
FragMan has was captured and gives way to another player
Dr. Death has been chosen to play. Welcome to your doom Dr. Death. >:)
HaPpyCaMpEr has been captured and gives way to another player
Igor has been chosen to play. Welcome to your doom Igor. >:)
Beatdown has been captured and gives way to another player
Executioner has been captured and gives way to another player
AxeMan has been chosen to play. Welcome to your doom AxeMan. >:)
BOOM has died of lead poisoning and gives way to another player
Thank you for playing, the game has ended.
Mr. Mxylplyx has died of lead poisoning and gives way to another player
```

12.6.3 讨论

信号量主要用于资源计数，并且命名信号量可用于跨进程使用（因为它们基于内核信号量对象）。对于许多 .NET 开发人员而言，在他们意识到*跨进程*（cross-process）也即意味着跨 AppDomain（cross-AppDomain）之前，可能不会对跨进程表现出太大的兴趣。如果你正在创建额外的 AppDomain 来包含你动态加载但不希望在自己的主 AppDomain 的整个生命期中都保留的程序集，那么信号量有助于你跟踪一次有多少资源被加载。控制一定数量的访问者在许多情景中都是很有用的（套接字编程、自定义线程池，等等）。

12.6.4 参考

MSDN 文档中的“Semaphore”“ManualResetEvent”和“ParameterizedThreadStart”主题。

12.7 使用互斥量同步多个进程

12.7.1 问题

你有两个进程或 AppDomain 在运行具有需要协调的行为的代码。

12.7.2 解决方案

要进行协调，可使用一个命名 Mutex 作为一个通用的信号机制。命名 Mutex 可以从运行于

不同进程或 AppDomain 的代码进行访问。

这样做很有用的一种情况是，你正在使用共享内存在进程间进行通信。本范例中展示的 SharedMemoryManager 类通过建立可用于在进程间传递可序列化对象的一段共享内存来展示实际使用的命名 Mutex。“服务器”进程创建一个 SharedMemoryManager 实例，它建立共享内存并作为最初的拥有者创建 Mutex。“客户端”进程然后同样创建了一个 SharedMemoryManager 实例，查找共享内存并与之连通。一旦连接建立，“客户端”进程就开始接收可序列化的对象并通过等待“服务器”进程创建的 Mutex 来等待一个对象发出。随后，“服务器”进程将一个可序列化对象序列化到共享内存中，并释放 Mutex。然后它再次等待，以便当“客户端”完成接收对象后，它能够释放 Mutex 并将控制权交还给“服务器”。在 Mutex 上等待的“客户端”进程然后反序列化来自共享内存的对象并释放 Mutex。

在示例中，你将发送如下的 Contact 结构。

```
[StructLayout(LayoutKind.Sequential)]
[Serializable()]
public struct Contact
{
    public string _name;
    public int _age;
}
```

发送 Contact 的“服务器”进程代码如下所示。

```
// 创建初始共享内存管理器以进行设置
using(SharedMemoryManager<Contact> sm =
    new SharedMemoryManager<Contact>("Contacts",8092))
{
    // 这是发送方进程

    // 启动第二个进程以继续
    string processName = Process.GetCurrentProcess().MainModule.FileName;
    int index = processName.IndexOf("vshost");
    if (index != -1)
    {
        string first = processName.Substring(0, index);
        int numChars = processName.Length - (index + 7);
        string second = processName.Substring(index + 7, numChars);

        processName = first + second;
    }
    Process receiver = Process.Start(
        new ProcessStartInfo(
            processName,
            "Receiver"));

    // 等待5秒
    Thread.Sleep(5000);

    // 创建一个contact
    Contact man;
    man._age = 23;
    man._name = "Dirk Daring";
```

```
        // 通过共享内存将其发送到其他进程
        sm.SendObject(man);
    }
```

接收 Contact 的“客户端”进程代码如下所示。

```
    // 创建初始共享内存管理器以进行设置
    using(SharedMemoryManager<Contact> sm =
        new SharedMemoryManager<Contact>("Contacts",8092))
    {

        // 一旦contact发送过来后获取它
        Contact c = (Contact)sm.ReceiveObject();

        // 将它写到控制台(或者到一个数据库……)
        Console.WriteLine("Contact {0} is {1} years old.",
                          c._name, c._age);

        // 等待5秒
        Thread.Sleep(5000);
    }
```

通常的工作方式是，一个进程使用 System.IO.MemoryMappedFile 创建一段分页文件支持的共享内存。你可以在例 12-7 中看到 MemoryMappedFile 在 SharedMemoryManager 的构造函数中建立。此构造函数接受一个名称作为共享内存的名称，以及要分配的共享内存块的基础大小。这是基础大小，因为 SharedMemoryManager 必须额外分配一点，以保持跟踪通过缓冲区移动的数据。

例 12-7：构造函数

```
    public SharedMemoryManager(string name,int sharedMemoryBaseSize)
    {
        // 只能够为可序列化的对象构建
        if (!typeof(TransferItemType).IsSerializable)
            throw new ArgumentException(
                $"Object {typeof(TransferItemType)} is not serializeable.");

        if (string.IsNullOrEmpty(name))
            throw new ArgumentNullException(nameof(name));

        if (sharedMemoryBaseSize <= 0)
            throw new ArgumentOutOfRangeException(nameof(sharedMemoryBaseSize),
                "Shared Memory Base Size must be a value greater than zero");

        // 设置区段的名称
        Name = name;

        // 保存基础大小
        SharedMemoryBaseSize = sharedMemoryBaseSize;

        // 建立共享内存区
        MemMappedFile = MemoryMappedFile.CreateOrOpen(Name, MemoryRegionSize);

        // 建立互斥量
```

```
        MutexForSharedMem = new Mutex(true, MutexName);
    }
```

通过共享内存发送一个对象的代码包含在 SendObject 方法中，如例 12-8 所示。首先，它检查要发送的对象是否可序列化，办法是通过检查对象类型上的 IsSerializable 属性。如果对象可序列化，则会将一个带有可序列化对象大小的整数和可序列化对象内容写到共享内存区。然后，Mutex 被释放，指示在共享内存中有一个对象。它随后再次等待 Mutex，直到"客户端"已经接收到这个对象为止。

例 12-8：SendObject 方法

```
    public void SendObject(TransferItemType transferObject)
    {
        // 创建内存流,初始大小
        using (MemoryStream ms = new MemoryStream())
        {
            // 获得一个用于序列化的格式化器
            BinaryFormatter formatter = new BinaryFormatter();
            try
            {
                // 将对象序列化到流中
                formatter.Serialize(ms, transferObject);

                // 获得序列化对象的字节
                byte[] bytes = ms.ToArray();

                // 检查该对象大小
                if(bytes.Length + sizeof(Int32)  > MemoryRegionSize)
                {
                    string msg =
                        $"{typeof(TransferItemType)} object instance serialized" +
                        $"to {bytes.Length} bytes which is too large for the shared " +
                        $"memory region";

                    throw new ArgumentException(msg, nameof(transferObject));
                }

                // 写入到共享内存区
                using (MemoryMappedViewStream stream =
                    MemMappedFile.CreateViewStream())
                {
                    BinaryWriter writer = new BinaryWriter(stream);
                    writer.Write(bytes.Length); // 写入大小
                    writer.Write(bytes); // 写入对象
                }
            }
            finally
            {
                // 通知其他使用互斥量的进程
                // 要进行接收处理
                MutexForSharedMem.ReleaseMutex();

                // 等待其他进程已完成接收的信号
                // 然后我们能够继续
                MutexForSharedMem.WaitOne();
```

```
        }
    }
}
```

例 12-9 中所示的 ReceiveObject 方法允许客户端等待直至共享内存中存在一个对象，然后它读取可序列化对象的大小，并将其序列化到一个托管对象。之后，它释放 Mutex，让发送方知道可以继续工作。

例 12-9：ReceiveObject 方法

```
public TransferItemType ReceiveObject()
{
    // 等待互斥量,直到发送方将一个对象放入队列
    MutexForSharedMem.WaitOne();

    // 从共享内存中获得对象
    byte[] serializedObj = null;
    using (MemoryMappedViewStream stream =
        MemMappedFile.CreateViewStream())
    {
        BinaryReader reader = new BinaryReader(stream);
        int objectLength = reader.ReadInt32();
        serializedObj = reader.ReadBytes(objectLength);
    }

    // 使用对象字节数据创建内存流
    using (MemoryStream ms = new MemoryStream(serializedObj))
    {
        // 建立一个二进制格式化器
        BinaryFormatter formatter = new BinaryFormatter();

        // 获得对象以返回
        TransferItemType item;
        try
        {
            item = (TransferItemType)formatter.Deserialize(ms);
        }
        finally
        {
            // 使用互斥量通知我们已接收到对象
            MutexForSharedMem.ReleaseMutex();
        }
        // 返回接收的对象
        return item;
    }
}
```

12.7.3 讨论

Mutex 设计用于给单个资源提供互斥访问（因此而得名）。Mutex 可被认为是一种跨进程的命名 Monitor，其中通过在 Mutex 上等待成为所有者而“进入”，通过为等待它的下一线程释放 Mutex 而“退出”。如果一个拥有 Mutex 的线程结束，那么 Mutex 会自动释放。

> 使用 Mutex 比使用 Monitor 要慢，因为 Monitor 是一种纯托管的构造，而 Mutex 则基于 Mutex 内核对象。Mutex 不能像 Monitor 那样是“脉冲式的”，但它可跨进程使用，而 Monitor 不能。最后，Mutex 基于 WaitHandle，所以它能够与其他从 WaitHandle 派生而来的对象一起被等待，例如 Semaphore 或事件类。

例 12-10 中完整地列出了 SharedMemoryManager 类。

例 12-10：SharedMemoryManager 类

```
/// <summary>
/// 通过共享内存来发送对象，
/// 使用互斥量来同步访问共享内存的类
/// </summary>
public class SharedMemoryManager<TransferItemType> : IDisposable
{
    #region Private members
    private bool disposed = false;
    #endregion

    #region Construction / Cleanup
    public SharedMemoryManager(string name,int sharedMemoryBaseSize)
    {
        // 只能够为可序列化的对象构建
        if (!typeof(TransferItemType).IsSerializable)
            throw new ArgumentException(
                $"Object {typeof(TransferItemType)} is not serializeable.");

        if (string.IsNullOrEmpty(name))
            throw new ArgumentNullException(nameof(name));

        if (sharedMemoryBaseSize <= 0)
            throw new ArgumentOutOfRangeException("sharedMemoryBaseSize",
                "Shared Memory Base Size must be a value greater than zero");

        // 设置区段的名称
        Name = name;

        // 保存基础大小
        SharedMemoryBaseSize = sharedMemoryBaseSize;

        // 建立共享内存区
        MemMappedFile = MemoryMappedFile.CreateOrOpen(Name, MemoryRegionSize);

        // 建立互斥量
        MutexForSharedMem = new Mutex(true, MutexName);
    }

    ~SharedMemoryManager()
    {
        // 确保关闭
        Dispose(false);
    }
```

```
public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}

private void Dispose(bool disposing)
{
    // 检查Dispose是否已被调用
    if (!this.disposed)
    {
        CloseSharedMemory();
    }
    disposed = true;
}

private void CloseSharedMemory()
{
    if(MemMappedFile != null)
        MemMappedFile.Dispose();
}

public void Close()
{
    CloseSharedMemory();
}
#endregion

#region Properties
/// <summary>
/// 内存映射文件拥有的大小
/// </summary>
public int SharedMemoryBaseSize { get; protected set; }

/// <summary>
/// 实际的内存区大小,
/// 以包含传输的对象的大小
/// </summary>
private long MemoryRegionSize => (long)(SharedMemoryBaseSize + sizeof(Int32));

/// <summary>
/// 共享内存区的名称
/// </summary>
private string Name { get; }

/// <summary>
/// 保护共享内存的互斥量的名称
/// </summary>
private string MutexName => $"{typeof(TransferItemType)}mtx{Name}";

/// <summary>
/// 保护共享内存的互斥量
/// </summary>
private Mutex MutexForSharedMem { get; } = null;
```

```
/// <summary>
/// 用于传输对象的MemoryMappedFile
/// </summary>
private MemoryMappedFile MemMappedFile { get; } = null;

#endregion

#region Public Methods
/// <summary>
/// 通过共享内存发送一个可序列化对象
/// 并等待对象被接收
/// </summary>
/// <param name="transferObject">要发送的对象</param>
public void SendObject(TransferItemType transferObject)
{
    // 创建内存流,初始大小
    using (MemoryStream ms = new MemoryStream())
    {
        // 获得一个用于序列化的格式化器
        BinaryFormatter formatter = new BinaryFormatter();
        try
        {
            // 将对象序列化到流中
            formatter.Serialize(ms, transferObject);

            // 获得序列化对象的字节
            byte[] bytes = ms.ToArray();

            // 检查该对象大小
            if(bytes.Length + sizeof(Int32)  > MemoryRegionSize)
            {
                string msg =
                    $"{typeof(TransferItemType)} object instance serialized" +
                    $"to {bytes.Length} bytes which is too large for the " +
                    $"shared memory region";

                throw new ArgumentException(msg, nameof(transferObject));
            }

            // 写入到共享内存区
            using (MemoryMappedViewStream stream =
                MemMappedFile.CreateViewStream())
            {
                BinaryWriter writer = new BinaryWriter(stream);
                writer.Write(bytes.Length); // 写入大小
                writer.Write(bytes); // 写入对象
            }
        }
        finally
        {
            // 通知其他使用互斥量的进程
            // 要进行接收处理
            MutexForSharedMem.ReleaseMutex();
```

```
                // wait for the other process to signal it has received
                // and we can move on
                MutexForSharedMem.WaitOne();
            }
        }
    }

    /// <summary>
    /// 等到一个对象进入共享内存,然后反序列化它
    /// </summary>
    /// <returns>传入的对象</returns>
    public TransferItemType ReceiveObject()
    {
        // 等待互斥量,直到发送方将一个对象放入队列
        MutexForSharedMem.WaitOne();

        // 从共享内存中获得对象
        byte[] serializedObj = null;
        using (MemoryMappedViewStream stream =
            MemMappedFile.CreateViewStream())
        {
            BinaryReader reader = new BinaryReader(stream);
            int objectLength = reader.ReadInt32();
            serializedObj = reader.ReadBytes(objectLength);
        }

        // 使用对象字节数据创建内存流
        using (MemoryStream ms = new MemoryStream(serializedObj))
        {
            // 建立一个二进制格式化器
            BinaryFormatter formatter = new BinaryFormatter();

            // 获得对象以返回
            TransferItemType item;
            try
            {
                item = (TransferItemType)formatter.Deserialize(ms);
            }
            finally
            {
                // 使用互斥量通知我们已接收到对象
                MutexForSharedMem.ReleaseMutex();
            }
            // 返回接收的对象
            return item;
        }
    }
    #endregion
}
```

12.7.4 参考

MSDN 文档中的“Memory-Mapped Files”“MemoryMappedFile 类”“Mutex”和“Mutex 类”主题。

12.8 使用事件协调线程

12.8.1 问题

你有多个需要由一台服务器来服务的线程，然而一次只能服务一个线程。

12.8.2 解决方案

当一个线程将要被服务时，使用 AutoResetEvent 通知每一个线程。例如，一个用餐者拥有一个厨师和多个服务员。服务员可以按顺序提供服务，但厨师每次只能服务一个人。你可以使用例 12-11 中给出的 Cook 类来对此进行模拟。

例 12-11：使用事件来使线程协作

```
public class Cook
{
    public string Name { get; set; }

    public static AutoResetEvent OrderReady =
        new AutoResetEvent(false);

    public void CallWaitress()
    {
        // 我们调用AutoResetEvent上的Set，
        // 但不需要像ManualResetEvent一样
        // 调用Reset以再次触发它
        // 这设置了服务员在GetInLine中等待的事件
        // 点餐准备好了……
        Console.WriteLine($"{Name} finished order!");
        OrderReady.Set();
    }
}
```

Cook 类有一个名为 OrderReady 的 AutoResetEvent，厨师用它来告知等待中的服务员，点餐已经准备好了。因为一次只能准备一份点餐，而且对于每位用餐者来说机会都是均等的，所以服务员将会首先为等待最久的客人送上点餐。当你调用 OrderReady 事件上的 Set 时，AutoResetEvent 只允许通知单个线程。

Waitress 类拥有由线程执行的 PlaceOrder 方法。PlaceOrder 接收两个参数，分别是服务员的姓名和 AutoResetEvent，然后调用 AutoResetEvent 上的 WaitOne 等待，直到点餐就绪为止。一旦 Cook 激发这个事件足够多次，使得服务员位于队列头，那么代码结束。

```
public class Waitress
{
    public static void PlaceOrder(string waitressName, AutoResetEvent orderReady)
    {
        // 下了点餐订单……
        Console.WriteLine($"Waitress {waitressName} placed order!");
        // 等餐……
        orderReady.WaitOne();
        // 点餐已做完……
```

```
            Console.WriteLine($"Waitress {waitressName} got order!");
        }
    }
```

运行“用餐者”的代码创建一个 Cook，并启动各个 Waitress 线程，然后在准备好饭菜后通过调用 AutoResetEvent 上的 Set 呼叫所有的服务员。

```
// 我们正在用餐,只有一个厨师,一次只能准备一份餐
Cook Mel = new Cook() { Name = "Mel" };
string[] waitressNames = { "Flo", "Alice", "Vera", "Jolene", "Belle" };

// 让服务员点好餐
foreach (var waitressName in waitressNames)
{
    Task.Run(() =>
        {
            // 服务员点好餐,然后等餐
            Waitress.PlaceOrder(waitressName, Cook.OrderReady);
        });
}

// 让厨师将点餐准备好
for (int i = 0; i < waitressNames.Length; i++)
{
    // 让服务员等一下……
    Thread.Sleep(2000);
    // ok,下一个服务员,送餐
    Mel.CallWaitress();
}
```

12.8.3 讨论

有两种事件类型存在：AutoResetEvent 和 ManualResetEvent。两类事件之间主要存在两个方面的不同。第一是 AutoResetEvent 只释放等待事件的众多线程中的一个，而 ManualResetEvent 在 Set 被调用时释放所有线程。第二点不同在于当在 AutoResetEvent 上调用 Set 时，它自动重置到一种无信号的状态，而 ManualResetEvent 会处于有信号的状态，直至调用 Reset 方法为止。

示例代码的输出如下所示。

```
Waitress Alice placed order!
Waitress Flo placed order!
Waitress Vera placed order!
Waitress Jolene placed order!
Mel finished order!
Waitress Alice got order!
Waitress Belle placed order!
Mel finished order!
Waitress Jolene got order!
Mel finished order!
Waitress Belle got order!
Mel finished order!
Waitress Flo got order!
```

```
Mel finished order!
Waitress Vera got order!
```

12.8.4　参考

MSDN 文档中的“AutoResetEvent”和“ManualResetEvent”主题，以及《Windows 核心编程》。

12.9　在多线程间执行原子操作

12.9.1　问题

你正在操作来自多个线程的数据，希望确保每次操作在执行来自另一线程的下一次操作之前完全执行。

12.9.2　解决方案

使用 Interlocked 系列函数来确保原子访问。Interlocked 拥有一些方法，用于增加和减少值，让一个给定的值增加一定数量，用一个新值替换一个初始值，比较当前值与初始值，以及当初始值等于当前值时将初始值替换为一个新值。

要增加和减少一个整数值，可分别使用 Increment 或 Decrement 方法。

```
int i = 0;
long l = 0;
Interlocked.Increment(ref i); // i = 1
Interlocked.Decrement(ref i); // i = 0
Interlocked.Increment(ref l); // l = 1
Interlocked.Decrement(ref i); // l = 0
```

要让一个给定的整数值增加特定数量，可使用 Add 方法。

```
Interlocked.Add(ref i, 10); // i = 10;
Interlocked.Add(ref l, 100); // l = 100;
```

要替换一个已有的值，可使用 Exchange 方法。

```
string name = "Mr. Ed";
Interlocked.Exchange(ref name, "Barney");
```

要在替换现有值之前检查是否另一个线程已经修改了来自现有代码的值，可使用 CompareExchange 方法。

```
int i = 0;
double runningTotal = 0.0;
double startingTotal = 0.0;
double calc = 0.0;
for (i = 0; i < 10; i++)
{
    do
```

```
    {
        // 保存初始值
        startingTotal = runningTotal;
        // 执行一个密集计算
        calc = runningTotal + i * Math.PI * 2 / Math.PI;
    }
    // 检查已确保runningTotal未被修改
    // 如果未修改,则使用calc替换
    // 如果已修改,则执行循环,直到我们获得了当前值为止
    while (startingTotal !=
        Interlocked.CompareExchange(
            ref runningTotal, calc, startingTotal));
}
```

12.9.3 讨论

对于诸如 Microsoft Windows 这种能够执行抢占式多任务的操作系统，工作于多线程时必须要考虑数据完整性。有许多同步原语可帮助保护代码段，以及在有可供修改的数据时发出信号。此列表还增加了一项能力，用于执行本质上是原子操作的那些操作。

如果你过去对于线程和汇编语言接触得不多，那么或许会想知道：这有什么大不了的，到底为什么需要这些原子函数？最基本的原因是，用 C# 编写的代码最终必须要被翻译为机器指令，而在这个过程中，用 C# 编写的一行代码能转换成多条指令由机器来执行。如果机器必须要执行多条指令来完成一个任务，而操作系统允许抢占，那么这些指令就可能没有作为一个单元被执行。在 C# 代码执行的过程中，它们会被其他代码中断，改变正被原先 C# 代码修改的值。可以想象，这可能会导致很严重的错误，或者就像对彩票号码进行四舍五入一样，让某个 C# 编程人员始终无法赢得大奖。

线程是一个强大的工具，但是与其他大多数“强大”的工具一样，你必须了解它的操作才能高效、安全地使用它。线程 bug 给调试带来很大的难题，因为运行时的行为并不是固定的。试图重现它们如同恶梦一般，添加日志会修改其行为，或者更糟——导致问题消失。要承认，在多线程环境中工作必须对保护数据访问进行一定的预先考虑，理解何时使用 `Interlocked` 类有助于避免长与调试器相伴的令人沮丧的漫漫长夜。

12.9.4 参考

MSDN 文档中的“`Interlocked`”和“`Interlocked` 类”主题。

12.10 优化以读为主的访问

12.10.1 问题

你在操作多数情况下读取只是偶尔会更新的数据，希望以线程安全但高效的方式执行这些操作。

12.10.2 解决方案

使用 ReaderWriterLockSlim，赋予多重读 / 单一写访问以及将锁从读升级为写的能力。举例来说，一个开发者开始做一个新项目。不幸的是，这个项目人员不足，所以，这个开发者必须响应来自团队中其他许多人员的任务。每个其他的团队成员也要求开发者更新他们任务上的状态，一些人甚至要改变开发者分配的任务的优先级。

通过 AddTask 方法向开发者分配一个任务。为了保护 DeveloperTasks 集合，我们使用一个 ReaderWriterLockSlim 上的写锁，在将任务添加到 DeveloperTasks 集合时调用 EnterWriteLock，添加完成之后调用 ExitWriteLock。

```
public void AddTask(DeveloperTask newTask)
{
    try
    {
        Lock.EnterWriteLock();
        // 如果我们已经有了此任务(名称唯一)
        // 那么就仅仅接受任务添加
        // 因为有时人们不止一次给你同一个任务
        var taskQuery = from t in DeveloperTasks
                        where t == newTask
                        select t;
        if (taskQuery.Count<DeveloperTask>() == 0)
        {
            Console.WriteLine($"Task {newTask.Name} was added to developer");
            DeveloperTasks.Add(newTask);
        }
    }
    finally
    {
        Lock.ExitWriteLock();
    }
}
```

当项目团队成员需要了解任务状态时，他们调用 IsTaskDone 方法。该方法通过调用 EnterReadLock 和 ExitReadLock 在 ReaderWriterLockSlim 上使用一个读锁。

```
public bool IsTaskDone(string taskName)
{
    try
    {
        Lock.EnterReadLock();
        var taskQuery = from t in DeveloperTasks
                        where t.Name == taskName
                        select t;
        if (taskQuery.Count<DeveloperTask>() > 0)
        {
            DeveloperTask task = taskQuery.First<DeveloperTask>();
            Console.WriteLine($"Task {task.Name} status was reported.");
            return task.Status;
        }
    }
    finally
```

```
        {
            Lock.ExitReadLock();
        }
        return false;
    }
```

开发团队的某些管理成员有权力提高他们赋予开发者的任务优先级。这可以通过调用开发者的 IncreasePriority 方法来完成。IncreasePriority 使用 ReaderWriterLockSlim 上的一个可升级锁，首先通过调用 EnterUpgradeableReadLock 方法获取一个读锁，然后如果任务在队列中，升级为一个写锁以调整任务的优先级。一旦优先级调整了，写锁被释放，它就会将锁降级到读锁，并通过调用 ExitUpgradeableReadLock 来释放锁。

```
    public void IncreasePriority(string taskName)
    {
        try
        {
            Lock.EnterUpgradeableReadLock();
            var taskQuery = from t in DeveloperTasks
                            where t.Name == taskName
                            select t;
            if(taskQuery.Count<DeveloperTask>()>0)
            {
                DeveloperTask task = taskQuery.First<DeveloperTask>();
                Lock.EnterWriteLock();
                task.Priority++;
                Console.WriteLine($"Task {task.Name}" +
                    $" priority was increased to {task.Priority}" +
                    " for developer");
                Lock.ExitWriteLock();
            }
        }
        finally
        {
            Lock.ExitUpgradeableReadLock();
        }
    }
```

12.10.3 讨论

出于以下三个原因，会创建 ReaderWriterLockSlim 以替代现有的 ReaderWriterLock。

- 使用 ReaderWriterLock 比使用 Monitor 慢 5 倍。
- ReaderWriterLock 的递归语义不标准，而且在某些线程重入的情况下会中断。
- ReaderWriterLock 的升级锁方法不是原子的。

ReaderWriterLockSlim 只比 Monitor 慢两倍，而且更加灵活、优先写，因此在写少读多的情况下，它比 Monitor 更具扩展性。同时还有方法可以确定持有何种锁并确定有多少线程正在等待获取它。

在默认情况下，不允许锁的递归获取。如果你调用 EnterReadLock 两次，将会得到一个 LockRecursionException。锁递归可以通过将 LockRecusionPolicy.SupportsRecursion 枚举

值传入重载的 ReaderWriterLockSlim 构造函数来启用。即使能够启用锁递归，一般也不推荐，因为它会让事情变得复杂，而且产生难以调试的问题。

> 存在以下这些不适合使用 ReaderWriterLockSlim 的情况，尽管其中大多数情况并不是日常开发中会遇到的。
>
> - 因为不兼容的 HostProtection 特性，所以 ReaderWriterLockSlim 不应在 SQL Server ClR 场景中使用。
> - 因为 ReaderWriterLockSlim 没有标记关键区域，使用线程中断的宿主不知道自己将会受到线程中断的何种损害，所以嵌入的 AppDomain 中会发生错误。
> - ReaderWriterLockSlim 不能处理异步异常（线程中断、内存溢出等），可能会在损坏的锁状态中结束。这可能导致死锁或者其他问题。
> - 因为 ReaderWriterLockSlim 具有线程关联性，所以它通常不能用于 async 和 await。对于这些情况，使用 SemaphoreSlim.WaitAsync 来代替。

针对这一示例的全部代码如下所示。

```
static Developer s_dev = null;
static bool s_end = false;

/// <summary>
/// </summary>
public static void TestReaderWriterLockSlim()
{
    s_dev = new Developer(15);
    LaunchTeam(s_dev);
    Thread.Sleep(10000);
}

private static void LaunchTeam(Developer dev)
{
    LaunchManager("CTO", dev);
    LaunchManager("Director", dev);
    LaunchManager("Project Manager", dev);
    LaunchDependent("Product Manager", dev);
    LaunchDependent("Test Engineer", dev);
    LaunchDependent("Technical Communications Professional", dev);
    LaunchDependent("Operations Staff", dev);
    LaunchDependent("Support Staff", dev);
}

public class DeveloperTaskInfo
{
    public string Name { get; set; }
    public Developer Developer { get; set; }
}

private static void LaunchManager(string name, Developer dev)
{
```

```
        var dti = new DeveloperTaskInfo() { Name = name, Developer = dev };
        Task manager = Task.Run(() => {
            Console.WriteLine($"Added {dti.Name} to the project...");
            DeveloperTaskManager mgr = new DeveloperTaskManager(dti.Name,
                dti.Developer);
        });
    }

    private static void LaunchDependent(string name, Developer dev)
    {
        var dti = new DeveloperTaskInfo() { Name = name, Developer = dev };
        Task manager = Task.Run(() => {
            Console.WriteLine($"Added {dti.Name} to the project...");
            DeveloperTaskDependent dep =
               new DeveloperTaskDependent(dti.Name, dti.Developer);
        });
    }

    public class DeveloperTask
    {
        public DeveloperTask(string name)
        {
            Name = name;
        }

        public string Name { get; set; }
        public int Priority { get; set; }
        public bool Status { get; set; }

        public override string ToString() => this.Name;

        public override bool Equals(object obj)
        {
            DeveloperTask task = obj as DeveloperTask;
            return this.Name == task?.Name;
        }

        public override int GetHashCode() => this.Name.GetHashCode();
    }

    public class Developer : IDisposable
    {
        /// <summary>
        /// 任务字典
        /// </summary>
        private List<DeveloperTask> DeveloperTasks { get; } =
            new List<DeveloperTask>();
        private ReaderWriterLockSlim Lock { get; set; } = new ReaderWriterLockSlim();
        private System.Threading.Timer Timer { get; set; }
        private int MaxTasks { get; }

        public Developer(int maxTasks)
        {
            // 开发者能接受而不至于离开的最大任务数
            MaxTasks = maxTasks;
```

```
        // 每1/4秒进行某项工作
        Timer = new Timer(new TimerCallback(DoWork), null, 1000, 250);
    }

    ~Developer()
    {
        Dispose(true);
    }

    // 执行一个任务
    protected void DoWork(Object stateInfo)
    {
        ExecuteTask();
        try
        {
            Lock.EnterWriteLock();
            // 如果完成了所有任务,就去度个假
            if (DeveloperTasks.Count == 0)
            {
                s_end = true;
                Console.WriteLine(
                    "Developer finished all tasks, go on vacation!");
                return;
            }

            if (!s_end)
            {
                // 如果有太多的任务,就退出
                if (DeveloperTasks.Count > MaxTasks)
                {
                    // 获得未完成的任务数量
                    var query = from t in DeveloperTasks
                                where t.Status == false
                                select t;
                    int unfinishedTaskCount = query.Count<DeveloperTask>();

                    s_end = true;
                    Console.WriteLine(
                        "Developer has too many tasks, quitting! " +
                        $"{unfinishedTaskCount} tasks left unfinished.");
                }
            }
            else
                Timer.Dispose();
        }
        finally
        {
            Lock.ExitWriteLock();
        }
    }

    public void AddTask(DeveloperTask newTask)
    {
        try
        {
```

```
            Lock.EnterWriteLock();
            // 如果我们已经有了此任务(名称唯一)
            // 那么就仅仅接受任务添加
            // 因为有时人们不止一次给你同一个任务
            var taskQuery = from t in DeveloperTasks
                            where t == newTask
                            select t;
            if (taskQuery.Count<DeveloperTask>() == 0)
            {
                Console.WriteLine($"Task {newTask.Name} was added to developer");
                DeveloperTasks.Add(newTask);
            }
        }
        finally
        {
            Lock.ExitWriteLock();
        }
    }

    /// <summary>
    /// 提高任务的优先级
    /// </summary>
    /// <param name="taskName">任务的名称</param>
    public void IncreasePriority(string taskName)
    {
        try
        {
            Lock.EnterUpgradeableReadLock();
            var taskQuery = from t in DeveloperTasks
                            where t.Name == taskName
                            select t;
            if(taskQuery.Count<DeveloperTask>()>0)
            {
                DeveloperTask task = taskQuery.First<DeveloperTask>();
                Lock.EnterWriteLock();
                task.Priority++;
                Console.WriteLine($"Task {task.Name}" +
                    $" priority was increased to {task.Priority}" +
                    " for developer");
                Lock.ExitWriteLock();
            }
        }
        finally
        {
            Lock.ExitUpgradeableReadLock();
        }
    }

    // <summary>
    // 允许人们检查任务是否已完成
    // </summary>
    // <param name="taskName">任务的名称</param>
    // <returns>如果任务未完成或不在列表中,则返回false
    // 如果已完成,则返回true</returns>
    public bool IsTaskDone(string taskName)
```

```
{
    try
    {
        Lock.EnterReadLock();
        var taskQuery = from t in DeveloperTasks
                        where t.Name == taskName
                        select t;
        if (taskQuery.Count<DeveloperTask>() > 0)
        {
            DeveloperTask task = taskQuery.First<DeveloperTask>();
            Console.WriteLine($"Task {task.Name} status was reported.");
            return task.Status;
        }
    }
    finally
    {
        Lock.ExitReadLock();
    }
    return false;
}

private void ExecuteTask()
{
    // 查看任务列表并执行最高优先级的任务
    var queryResult =   from t in DeveloperTasks
                        where t.Status == false
                        orderby t.Priority
                        select t;
    if (queryResult.Count<DeveloperTask>() > 0)
    {
        // 执行任务
        DeveloperTask task = queryResult.First<DeveloperTask>();
        task.Status = true;
        task.Priority = -1;
        Console.WriteLine($"Task {task.Name} executed by developer.");
    }
}

#region IDisposable Support
private bool disposedValue = false; // 用于检测冗余的调用

protected virtual void Dispose(bool disposing)
{
    if (!disposedValue)
    {
        if (disposing)
        {
            Lock?.Dispose();
            Lock = null;
            Timer?.Dispose();
            Timer = null;
        }
        disposedValue = true;
    }
}
```

```
    public void Dispose()
    {
        Dispose(true);
    }
    #endregion
}

public class DeveloperTaskManager : DeveloperTaskDependent, IDisposable
{
    private System.Threading.Timer ManagerTimer { get; set; }

    public DeveloperTaskManager(string name, Developer taskExecutor) :
        base(name, taskExecutor)
    {
        // 每2秒进行干预
        ManagerTimer =
            new Timer(new TimerCallback(Intervene), null, 0, 2000);
    }

    ~DeveloperTaskManager()
    {
        Dispose(true);
    }

    // 干预任务
    protected void Intervene(Object stateInfo)
    {
        ChangePriority();
        // 开发完成,释放计时器
        if (s_end)
        {
            ManagerTimer.Dispose();
            TaskExecutor = null;
        }
    }

    public void ChangePriority()
    {
        if (DeveloperTasks.Count > 0)
        {
            int taskIndex = _rnd.Next(0, DeveloperTasks.Count - 1);
            DeveloperTask checkTask = DeveloperTasks[taskIndex];
            // 确认开发者在某些随机的任务上干快一些
            if (TaskExecutor != null)
            {
                TaskExecutor.IncreasePriority(checkTask.Name);
                Console.WriteLine(
                    $"{Name} intervened and changed priority for task
{checkTask.Name}");
            }
        }
    }

    #region IDisposable Support
```

```
    private bool disposedValue = false; // 用于检测冗余的调用

    protected override void Dispose(bool disposing)
    {
        if (!disposedValue)
        {
            if (disposing)
            {
                ManagerTimer?.Dispose();
                ManagerTimer = null;
                base.Dispose(disposing);
            }
            disposedValue = true;
        }
    }

    public new void Dispose()
    {
        Dispose(true);
    }
    #endregion
}

public class DeveloperTaskDependent : IDisposable
{
    protected List<DeveloperTask> DeveloperTasks { get; set; }
        = new List<DeveloperTask>();
    protected Developer TaskExecutor { get; set; }

    protected Random _rnd = new Random();
    private Timer TaskTimer { get; set; }
    private Timer StatusTimer { get; set; }

    public DeveloperTaskDependent(string name, Developer taskExecutor)
    {
        Name = name;
        TaskExecutor = taskExecutor;
        // 每秒钟添加工作
        TaskTimer = new Timer(new TimerCallback(AddWork), null, 0, 1000);
        // 每3秒检查状态
        StatusTimer = new Timer(new TimerCallback(CheckStatus), null, 0, 3000);
    }

    ~DeveloperTaskDependent()
    {
        Dispose();
    }

    // 添加更多的工作给开发者
    protected void AddWork(Object stateInfo)
    {
        SubmitTask();
        // 开发完成,释放计时器
        if (s_end)
        {
```

```
            TaskTimer.Dispose();
            TaskExecutor = null;
        }
    }

    // 检查开发者的工作状态
    protected void CheckStatus(Object stateInfo)
    {
        CheckTaskStatus();
        // 开发完成,释放计时器
        if (s_end)
        {
            StatusTimer.Dispose();
            TaskExecutor = null;
        }
    }

    public string Name { get; set; }

    public void SubmitTask()
    {
        int taskId = _rnd.Next(10000);
        string taskName = $"({taskId} for {Name})";
        DeveloperTask newTask = new DeveloperTask(taskName);

        if (TaskExecutor != null)
        {
            TaskExecutor.AddTask(newTask);
            DeveloperTasks.Add(newTask);
        }
    }

    public void CheckTaskStatus()
    {
        if (DeveloperTasks.Count > 0)
        {
            int taskIndex = _rnd.Next(0, DeveloperTasks.Count - 1);
            DeveloperTask checkTask = DeveloperTasks[taskIndex];
            if (TaskExecutor != null &&
                TaskExecutor.IsTaskDone(checkTask.Name))
            {
                Console.WriteLine($"Task {checkTask.Name} is done for {Name}");
                // 从待办列表中移除它
                DeveloperTasks.Remove(checkTask);
            }
        }
    }

    #region IDisposable Support
    private bool disposedValue = false; // 用于检测冗余的调用

    protected virtual void Dispose(bool disposing)
    {
        if (!disposedValue)
        {
```

```
            if (disposing)
            {
                TaskTimer?.Dispose();
                TaskTimer = null;
                StatusTimer?.Dispose();
                StatusTimer = null;
            }
            disposedValue = true;
        }
    }

    public void Dispose()
    {
        Dispose(true);
    }
    #endregion
}
```

你可以在输出中看到项目的事件序列。

```
Added CTO to the project...
Added Director to the project...
Added Project Manager to the project...
Added Product Manager to the project...
Added Test Engineer to the project...
Added Technical Communications Professional to the project...
Added Operations Staff to the project...
Added Support Staff to the project...
Task (6267 for CTO) was added to developer
Task (6267 for CTO) status was reported.
Task (6267 for CTO) priority was increased to 1 for developer
CTO intervened and changed priority for task (6267 for CTO)
Task (6267 for Director) was added to developer
Task (6267 for Director) status was reported.
Task (6267 for Director) priority was increased to 1 for developer
Director intervened and changed priority for task (6267 for Director)
Task (6267 for Project Manager) was added to developer
Task (6267 for Project Manager) status was reported.
Task (6267 for Project Manager) priority was increased to 1 for developer
Project Manager intervened and changed priority for task (6267 for Project
Manager)
Task (6267 for Product Manager) was added to developer
Task (6267 for Product Manager) status was reported.
Task (6267 for Technical Communications Professional) was added to developer
Task (6267 for Technical Communications Professional) status was reported.
Task (6267 for Operations Staff) was added to developer
Task (6267 for Operations Staff) status was reported.
Task (6267 for Support Staff) was added to developer
Task (6267 for Support Staff) status was reported.
Task (6267 for Test Engineer) was added to developer
Task (5368 for CTO) was added to developer
Task (5368 for Director) was added to developer
Task (5368 for Project Manager) was added to developer
Task (6153 for Product Manager) was added to developer
Task (913 for Test Engineer) was added to developer
```

```
Task (6153 for Technical Communications Professional) was added to developer
Task (6153 for Operations Staff) was added to developer
Task (6153 for Support Staff) was added to developer
Task (6267 for Product Manager) executed by developer.
Task (6267 for Technical Communications Professional) executed by developer.
Task (6267 for Operations Staff) executed by developer.
Task (6267 for Support Staff) executed by developer.
Task (6267 for CTO) priority was increased to 2 for developer
CTO intervened and changed priority for task (6267 for CTO)
Task (6267 for Director) priority was increased to 2 for developer
Director intervened and changed priority for task (6267 for Director)
Task (6267 for Project Manager) priority was increased to 2 for developer
Project Manager intervened and changed priority for task (6267 for Project
Manager)
Task (6267 for Test Engineer) executed by developer.
Task (7167 for CTO) was added to developer
Task (7167 for Director) was added to developer
Task (7167 for Project Manager) was added to developer
Task (5368 for Product Manager) was added to developer
Task (6153 for Test Engineer) was added to developer
Task (5368 for Technical Communications Professional) was added to developer
Task (5368 for Operations Staff) was added to developer
Task (5368 for Support Staff) was added to developer
Task (5368 for CTO) executed by developer.
Task (5368 for Director) executed by developer.
Task (5368 for Project Manager) executed by developer.
Task (6267 for CTO) status was reported.
Task (6267 for Director) status was reported.
Task (6267 for Project Manager) status was reported.
Task (913 for Test Engineer) status was reported.
Task (6267 for Technical Communications Professional) status was reported.
Task (6267 for Technical Communications Professional) is done for Technical
Communications Professional
Task (6267 for Product Manager) status was reported.
Task (6267 for Product Manager) is done for Product Manager
Task (6267 for Operations Staff) status was reported.
Task (6267 for Operations Staff) is done for Operations Staff
Task (6267 for Support Staff) status was reported.
Task (6267 for Support Staff) is done for Support Staff
Task (6153 for Product Manager) executed by developer.
Task (2987 for CTO) was added to developer
Task (2987 for Director) was added to developer
Task (2987 for Project Manager) was added to developer
Task (7167 for Product Manager) was added to developer
Task (4126 for Test Engineer) was added to developer
Task (7167 for Technical Communications Professional) was added to developer
Task (7167 for Support Staff) was added to developer
Task (7167 for Operations Staff) was added to developer
Task (913 for Test Engineer) executed by developer.
Task (6153 for Technical Communications Professional) executed by developer.
Developer has too many tasks, quitting! 21 tasks left unfinished.
Task (6153 for Operations Staff) executed by developer.
Task (5368 for CTO) priority was increased to 0 for developer
CTO intervened and changed priority for task (5368 for CTO)
Task (5368 for Director) priority was increased to 0 for developer
```

```
Director intervened and changed priority for task (5368 for Director)
Task (5368 for Project Manager) priority was increased to 0 for developer
Project Manager intervened and changed priority for task (5368 for Project
Manager)
Task (6153 for Support Staff) executed by developer.
Task (4906 for Product Manager) was added to developer
Task (7167 for Test Engineer) was added to developer
Task (4906 for Technical Communications Professional) was added to developer
Task (4906 for Operations Staff) was added to developer
Task (4906 for Support Staff) was added to developer
Task (7167 for CTO) executed by developer.
Task (7167 for Director) executed by developer.
Task (7167 for Project Manager) executed by developer.
Task (5368 for Product Manager) executed by developer.
Task (6153 for Test Engineer) executed by developer.
Task (5368 for Technical Communications Professional) executed by developer.
Task (5368 for Operations Staff) executed by developer.
Task (5368 for Support Staff) executed by developer.
Task (2987 for CTO) executed by developer.
Task (2987 for Director) executed by developer.
Task (2987 for Project Manager) executed by developer.
Task (7167 for Product Manager) executed by developer.
Task (4126 for Test Engineer) executed by developer.
```

12.10.4　参考

MSDN 文档中的“`ReaderWriterLockSlim`”主题。

12.11　使数据库请求更具扩展性

12.11.1　问题

你希望使你的数据库调用从调用方的角度来看尽可能高效和可扩展。

12.11.2　解决方案

使用 `async`、`await` 和 `*Async` 版本的数据库调用。这样在等待数据库 I/O 完成时，应用程序中的线程可以完成其他工作。

> 如果你不熟悉在 C# 5.0 中引入的 `async` 和 `await`，请参阅 MSDN 主题“使用 `Async` 和 `Await` 的异步编程”以了解详细信息。

如果你在用 `SqlConnection` 和 `SqlCommand`，可以使用 `SqlConnection.OpenAsync` 和 `SqlCommand.ExecuteReaderAsync` 方法以异步方式打开并查询数据库，代码如下所示。

```
using (SqlConnection conn =
    new SqlConnection(Settings.Default.NorthwindConnectionString))
{
    await conn.OpenAsync();
    SqlCommand cmd = new SqlCommand("SELECT * FROM CUSTOMERS", conn);
    SqlDataReader reader = await cmd.ExecuteReaderAsync();
    while (reader.Read())
    {
        Console.WriteLine($"Customer {reader["ContactName"].ToString()} " +
            $"from {reader["CompanyName"].ToString()}");
    }
}
```

如果你在用 Entity Framework，可以使用 IQueryable<T>.ToListAsync 扩展方法以异步方式打开连接并执行查询，代码如下所示。

```
using (var efContext = new NorthwindEntities())
{
    var list = await (from cust in efContext.Customers
                select cust).ToListAsync();

    foreach(var cust in list)
    {
        Console.WriteLine($"Customer {cust.ContactName} " +
            $"from {cust.CompanyName}");
    }
```

如果你想使用 EntityFramework 写一条新的记录，然后获取它以查看，可以在将新实体添加到上下文之后使用 System.Data.Entity.DbContext.SaveChangesAsync 方法。然后你可以使用 IQueryable<T>.FirstOrDefaultAsync 扩展方法获取第一个匹配项或 null，代码如下所示。

```
    // 创建新的customer并且保存
    Customer c = new Customer();
    c.CustomerID = "JENNA";
    c.ContactName = "Jenna Roberts";
    c.CompanyName = "Flamingo Industries";
    efContext.Customers.Add(c);
    await efContext.SaveChangesAsync();

    var jenna = await efContext.Customers.Where(cu =>
                    cu.ContactName == "Jenna Roberts").FirstOrDefaultAsync();
    Console.WriteLine($"New Customer {jenna.ContactName} " +
            $"from {jenna.CompanyName}");
}
```

12.11.3 讨论

尽管一些数据库技术目前已实现异步支持，但并不是所有数据库技术都这么幸运。LINQ to SQL 仍然派生自 System.Data.Linq.DataContext，它没有异步支持。不过你仍可以将 LINQ to SQL 用于非异步操作，如下所示。

```
using (var l2sContext = new NorthwindLinq2SqlDataContext())
{
    var list = (from cust in l2sContext.Customers
                select cust);
    foreach (var cust in list)
    {
        Console.WriteLine($"Customer {cust.ContactName} " +
            $"from {cust.CompanyName}");
    }
}
```

如果你尝试将如下所示的 async 支持与 LINQ to SQL 一起使用，将收到此错误。

```
    var list = await (from cust in l2sContext.Customers
                      select cust).ToListAsync();
// 额外信息: The source IQueryable doesn't implement
// IDbAsyncEnumerable<NorthwindLinq2Sql.Customer>. Only sources that implement
// IDbAsyncEnumerable can be used for Entity Framework
```

如果你想要数据库操作的异步支持，需要使用 System.Data.SqlClient 构造（如 SqlConnection、SqlDataReader 等）或使用 Entity Framework 6 或更高版本。

12.11.4 参考

MSDN 文档中的“System.Data.SqlClient 命名空间”“使用 Async 和 Await 的异步编程”和“Entity Framework Async Query & Save”主题。

12.12 以一定顺序运行任务

12.12.1 问题

在你的应用程序中，你有需要在从属任务执行之前完成的初始任务。

12.12.2 解决方案

使用 Task.ContinueWith 以在初始任务完成之后执行后续任务。

ContinueWith 允许你追加一个任务在初始任务完成之后异步执行。这在一些任务有顺序约束而另一些任务没有顺序约束的情况下很有用。

作为一个例子，想想奥运会 4 × 400 米接力赛。其中有一些初级任务（每个国家的第一棒起跑者），接下来是依赖第一个任务结果的任务（跑接力剩余棒的选手）。在前一个任务完成（接力棒传递）之前，这些依赖任务都不能开始。

要表示每个接力中的选手，我们有一个 RelayRunner 类，包含选手代表的国家（Country），其所跑的那一段（Leg），目前是否拥有接力棒（HasBaton），以及跑完接力这一段所花的时间（LegTime）。最后，有一个方法，使 RelayRunner 在轮到该选手时冲刺（Sprint）。

```
public class RelayRunner
{
```

```
        public string Country { get; set; }
        public int Leg { get; set; }
        public bool HasBaton { get; set; }
        public TimeSpan LegTime { get; set; }
        public int TotalLegs { get; set; }

        public RelayRunner Sprint()
        {
            Console.WriteLine(
                $"{Country} for Leg {Leg} has the baton and is running!");
            Random rnd = new Random();
            int ms = rnd.Next(100, 1000);
            Task.Delay(ms);
            // 结束了……
            LegTime = new TimeSpan(0,0,0,0,ms);
            if (Leg == TotalLegs)
                Console.WriteLine($"{Country} has finished the race!");
            return this;
        }
    }
```

有了选手之后，我们需要建立他们代表的国家（countries）和一些跟踪变量，关于每个团队中有谁参加赛跑（teams），是谁在参加比赛（runners），以及谁是第一棒起跑者（firstLegRunners）。

```
    // 奥运会中的接力跑
    string[] countries = { "Russia", "France", "England", "United States",
                           "India", "Germany", "China" };
    Task<RelayRunner>[,] teams = new Task<RelayRunner>[countries.Length, 4];
    List<Task<RelayRunner>> runners = new List<Task<RelayRunner>>();
    List<Task<RelayRunner>> firstLegRunners = new List<Task<RelayRunner>>();
```

我们将生成这些集合和选手，这样每个团队的第一棒起跑者都有接力棒；如果选手不是团队中的起跑者，则其起步取决于团队中的前一个选手何时完成（ContinueWith），代码如下所示。

```
    for (int i = 0; i < countries.Length; i++)
    {
        for (int r = 0; r < 4; r++)
        {
            var runner = new RelayRunner()
            {
                Country = countries[i],
                Leg = r+1,
                HasBaton = r == 0 ? true : false,
                TotalLegs = 4
            };

            if (r == 0) // 为每个国家添加起跑者
            {
                Func<RelayRunner> funcRunner = runner.Sprint;
                teams[i, r] = new Task<RelayRunner>(funcRunner);
                firstLegRunners.Add(teams[i, r]);
            }
```

```
            else // 添加每个国家的其他接力段的选手
            {
                teams[i, r] = teams[i, r - 1].ContinueWith((lastRunnerRunning) =>
                    {
                        var lastRunner = lastRunnerRunning.Result;
                        // 接棒
                        Console.WriteLine($"{lastRunner.Country} hands off from " +
                            $"{lastRunner.Leg} to {runner.Leg}!");
                        Random rnd = new Random();
                        int fumbleChance = rnd.Next(0, 10);
                        if (fumbleChance > 8)
                        {
                            Console.WriteLine(
                                $"Oh no! {lastRunner.Country} for Leg " +
                                $"{runner.Leg} fumbled the hand off from Leg " +
                                $"{lastRunner.Leg}!");
                            Thread.Sleep(1000);
                            Console.WriteLine($"{lastRunner.Country} for Leg " +
                                $"{runner.Leg}" +
                                " recovered the baton and is running again!");
                        }
                        lastRunner.HasBaton = false;
                        runner.HasBaton = true;
                        return runner.Sprint();
                    });
            }
            // 添加到我们的runners列表中
            runners.Add(teams[i, r]);
        }
    }
```

要模拟发令枪，我们将使用 Parallel.ForEach 来调用每个第一棒起跑者任务的 Start。与通过简单循环相比，这样确保了更多的随机启动，代码如下所示。

```
    // 发令枪响以启动赛跑
    Parallel.ForEach(firstLegRunners, r =>
    {
        r.Start();
    });
```

最后，我们使用所有 Task<RelayRunner> 任务的列表调用 Task.WaitAll 以等待比赛结束，代码如下所示。

```
    // 等待每个人完成
    Task.WaitAll(runners.ToArray());
```

12.12.3 讨论

尽管以一定顺序运行任务通常来说违反了并行性，因为能够以任意顺序运行不相关的任务对可扩展性最好，但现实是大多数任务是有顺序的，并且能够在代码中表示该顺序是有用的。ContinueWith 提供了一些带有不同参数的重载以控制在初始任务完成之后发生的行为。表 12-1 列出了作为参数可用的控制结构类型。

表12-1：ContinueWith参数

值	描　　述
Action<Task>	初始任务完成之后执行的动作。将会把原始任务作为引用传入
Func<Task, TResult>	初始任务完成之后执行的函数。将会把原始任务作为引用传入
CancellationToken	此标记将赋给延续任务以允许取消延续
TaskScheduler	此任务使用的调度器（与默认不同，如果基于任务需要不同的调度算法）
Object	传入到延续中的状态对象

若要在比赛结束之后显示积分榜，我们使用下面的代码，其中对选手按团队分组并使用 LINQ 中的 Sum 扩展方法累加他们的时间（LegTime）。

```
Console.WriteLine("\r\nRace standings:");

var standings = from r in runners
                group r by r.Result.Country into countryTeams
                select countryTeams;

string winningCountry = string.Empty;
int bestTime = int.MaxValue;

HashSet<Tuple<int, string>> place = new HashSet<Tuple<int, string>>();
foreach (var team in standings)
{
    var time = team.Sum(r => r.Result.LegTime.Milliseconds);
    if (time < bestTime)
    {
        bestTime = time;
        winningCountry = team.Key;
    }
    place.Add(new Tuple<int, string>(time,
        $"{team.Key} with a time of {time}ms"));
}
int p = 1;
foreach(var item in place.OrderBy(t => t.Item1))
{
    Console.WriteLine($"{p}: {item.Item2}");
    p++;
}
Console.WriteLine($"\n\nThe winning team is from {winningCountry}");
```

比赛的输出如下所示。

```
France for Leg 1 has the baton and is running!
United States for Leg 1 has the baton and is running!
Russia for Leg 1 has the baton and is running!
England for Leg 1 has the baton and is running!
France hands off from 1 to 2!
England hands off from 1 to 2!
Russia hands off from 1 to 2!
United States hands off from 1 to 2!
Russia for Leg 2 has the baton and is running!
Oh no! England for Leg 2 fumbled the hand off from Leg 1!
```

```
Oh no! France for Leg 2 fumbled the hand off from Leg 1!
United States for Leg 2 has the baton and is running!
Russia hands off from 2 to 3!
United States hands off from 2 to 3!
Russia for Leg 3 has the baton and is running!
Russia hands off from 3 to 4!
United States for Leg 3 has the baton and is running!
Russia for Leg 4 has the baton and is running!
United States hands off from 3 to 4!
United States for Leg 4 has the baton and is running!
United States has finished the race!
Russia has finished the race!
Germany for Leg 1 has the baton and is running!
Germany hands off from 1 to 2!
Germany for Leg 2 has the baton and is running!
Germany hands off from 2 to 3!
Germany for Leg 3 has the baton and is running!
India for Leg 1 has the baton and is running!
India hands off from 1 to 2!
India for Leg 2 has the baton and is running!
Germany hands off from 3 to 4!
Germany for Leg 4 has the baton and is running!
India hands off from 2 to 3!
India for Leg 3 has the baton and is running!
India hands off from 3 to 4!
India for Leg 4 has the baton and is running!
India has finished the race!
China for Leg 1 has the baton and is running!
Germany has finished the race!
China hands off from 1 to 2!
China for Leg 2 has the baton and is running!
China hands off from 2 to 3!
China for Leg 3 has the baton and is running!
China hands off from 3 to 4!
China for Leg 4 has the baton and is running!
China has finished the race!
France for Leg 2 recovered the baton and is running again!
France for Leg 2 has the baton and is running!
France hands off from 2 to 3!
France for Leg 3 has the baton and is running!
France hands off from 3 to 4!
France for Leg 4 has the baton and is running!
France has finished the race!
England for Leg 2 recovered the baton and is running again!
England for Leg 2 has the baton and is running!
England hands off from 2 to 3!
England for Leg 3 has the baton and is running!
England hands off from 3 to 4!
England for Leg 4 has the baton and is running!
England has finished the race!

Race standings:
1: India with a time of 696ms
2: Germany with a time of 698ms
3: China with a time of 699ms
```

```
4: Russia with a time of 1510ms
5: United States with a time of 1540ms
6: France with a time of 2659ms
7: England with a time of 3625ms


The winning team is from India
```

12.12.4 参考

MSDN 文档中的“`Task.ContinueWith`”“`Task.WaitAll`”和“`Parallel.ForEach`”主题，以及 Stephen Cleary 著的《C# 并发编程经典实例》。

第 13 章

工具箱

13.0 简介

每个程序员都有一些不断参考并使用的例程。这些工具函数通常是一些未由任何特定语言或框架提供的代码。本章汇编了我们在使用 C# 和 .NET Framework 过程中收集的实用工具例程。我们在本章中共享的各类内容包括以下这些。

- 电源管理事件
- 确定操作系统中各个位置的路径
- 与服务交互
- 检查全局程序集缓存（GAC）
- 消息队列

这是你在开发自己应用程序中的大量功能集时帮助解决特定需求的各种代码汇集。

13.1 处理操作系统关机、电源管理或用户会话变化

13.1.1 问题

你希望在操作系统或用户启动某个要求你的应用程序关闭或停止的操作（用户注销、远程会话断开、系统关机、休眠 / 恢复，等等）时得到通知。有了这个通知之后，你可以使你的应用程序优雅地响应变化。

13.1.2　解决方案

使用 `Microsoft.Win32.SystemEvents` 类获得操作系统、用户会话修改以及电源管理事件的通知。接下来所示的 `RegisterForSystemEvents` 方法挂接五个捕获这些事件必需的事件处理程序，并且应该将该方法放置在代码的初始化部分中。

```
public static void RegisterForSystemEvents()
{
    // 始终在事件线程关闭时得到最终通知以便注销
    SystemEvents.EventsThreadShutdown +=
        new EventHandler(OnEventsThreadShutdown);
    SystemEvents.PowerModeChanged +=
        new PowerModeChangedEventHandler(OnPowerModeChanged);
    SystemEvents.SessionSwitch +=
        new SessionSwitchEventHandler(OnSessionSwitch);
    SystemEvents.SessionEnding +=
        new SessionEndingEventHandler(OnSessionEnding);
    SystemEvents.SessionEnded +=
        new SessionEndedEventHandler(OnSessionEnded);
}
```

`EventsThreadShutdown` 事件在分发来自 `SystemEvents` 类的事件的线程关闭时通知你，因此你可以在 `SystemEvents` 类上注销尚未注销的事件。`PowerModeChanged` 事件在用户挂起系统或从被挂起状态恢复系统时触发。`SessionSwitch` 事件在登录用户变化时触发。当用户试图登出或关闭系统时触发 `SessionEnding` 事件，而 `SessionEnded` 事件在用户实际登出或关闭系统时触发。

可以使用 `UnregisterFromSystemEvents` 方法注销事件。`UnregisterFromSystemEvents` 应当在 Windows Forms、用户控件或者任何其他来来往往的类终止代码中调用，以及在范例 13.2（即 13.2 节）中显示的另一个区域中调用。

```
private static void UnregisterFromSystemEvents()
{
    SystemEvents.EventsThreadShutdown -=
        new EventHandler(OnEventsThreadShutdown);
    SystemEvents.PowerModeChanged -=
        new PowerModeChangedEventHandler(OnPowerModeChanged);
    SystemEvents.SessionSwitch -=
        new SessionSwitchEventHandler(OnSessionSwitch);
    SystemEvents.SessionEnding -=
        new SessionEndingEventHandler(OnSessionEnding);
    SystemEvents.SessionEnded -=
        new SessionEndedEventHandler(OnSessionEnded);
}
```

> 由于 `SystemEvents` 公开的事件是静态的，如果你在一段可能被多次调用的代码中使用它们（次级 Windows Forms、用户控件、监控类等），那么必须注销自己的处理程序，否则将会导致应用程序的内存泄漏。

`SystemEvents` 事件处理程序方法是用于每个已在 `RegisterForSystemEvents` 中订阅的事件

的独立处理程序。要介绍的第一个处理程序是 OnEventsThreadShutdown 处理程序。如果该事件被激发，那么注销处理程序是很重要的，因为用于 SystemEvents 类的通知终程正在停止，而且该类可能在你的应用程序之前结束。如果在该时间点之前不注销，将会导致内存泄漏。因此，要在此事件处理程序中添加对 UnregisterFromSystemEvents 的调用，代码如下所示。

```
private static void OnEventsThreadShutdown(object sender, EventArgs e)
{
    Debug.WriteLine(
        "System event thread is shutting down, no more notifications.");

    // 注销所有事件处理,因为通知线程就要结束
    UnregisterFromSystemEvents();
}
```

下一个要探索的处理程序是 OnPowerModeChanged 方法。该处理程序可以通过 PowerModeChangedEventArgs 参数的 Mode 属性报告电源管理事件的类型。Mode 属性具有 PowerMode 枚举类型，通过其包含的枚举值指定事件类型。

```
private static void OnPowerModeChanged(object sender,
    PowerModeChangedEventArgs e)
{
    // 电源模式在变化
    switch (e.Mode)
    {
        case PowerModes.Resume:
            Debug.WriteLine("PowerMode: OS is resuming from suspended state");
            break;
        case PowerModes.StatusChange:
            Debug.WriteLine(
                "PowerMode: There was a change relating to the power" +
                " supply (weak battery, unplug, etc..)");
            break;
        case PowerModes.Suspend:
            Debug.WriteLine("PowerMode: OS is about to be suspended");
            break;
    }
}
```

接下来的三个处理程序用来处理操作系统会话状态，它们是 OnSessionSwitch、OnSessionEnding 和 OnSessionEnded。处理这三种事件覆盖了应用程序可能需要关心的所有操作系统会话状态转变。在 OnSessionEnding 中有一个 SessionEndingEventArgs 参数，它有一个 Cancel 成员。通过将 Cancel 成员设置为 false，允许你请求会话不要终止。用于这三个处理程序的代码如例 13-1 所示。

例 13-1：OnSessionSwitch、OnSessionEnding 和 OnSessionEnded 处理程序

```
private static void OnSessionSwitch(object sender, SessionSwitchEventArgs e)
{
    // 检查原因
    switch (e.Reason)
    {
        case SessionSwitchReason.ConsoleConnect:
```

```
                Debug.WriteLine("Session connected from the console");
                break;
            case SessionSwitchReason.ConsoleDisconnect:
                Debug.WriteLine("Session disconnected from the console");
                break;
            case SessionSwitchReason.RemoteConnect:
                Debug.WriteLine("Remote session connected");
                break;
            case SessionSwitchReason.RemoteDisconnect:
                Debug.WriteLine("Remote session disconnected");
                break;
            case SessionSwitchReason.SessionLock:
                Debug.WriteLine("Session has been locked");
                break;
            case SessionSwitchReason.SessionLogoff:
                Debug.WriteLine("User was logged off from a session");
                break;
            case SessionSwitchReason.SessionLogon:
                Debug.WriteLine("User has logged on to a session");
                break;
            case SessionSwitchReason.SessionRemoteControl:
                Debug.WriteLine("Session changed to or from remote status");
                break;
            case SessionSwitchReason.SessionUnlock:
                Debug.WriteLine("Session has been unlocked");
                break;
        }
    }

    private static void OnSessionEnding(object sender, SessionEndingEventArgs e)
    {
        // 设为true以取消结束会话的用户请求,否则设为false
        e.Cancel = false;
        // 检查原因
        switch(e.Reason)
        {
            case SessionEndReasons.Logoff:
                Debug.WriteLine("Session ending as the user is logging off");
                break;
            case SessionEndReasons.SystemShutdown:
                Debug.WriteLine("Session ending as the OS is shutting down");
                break;
        }
    }

    private static void OnSessionEnded(object sender, SessionEndedEventArgs e)
    {
        switch (e.Reason)
        {
            case SessionEndReasons.Logoff:
                Debug.WriteLine("Session ended as the user is logging off");
                break;
            case SessionEndReasons.SystemShutdown:
                Debug.WriteLine("Session ended as the OS is shutting down");
                break;
```

```
        }
    }
```

13.1.3 讨论

.NET Framework 提供了当用户或系统交互引起改变时从系统中获得反馈的机会。SystemEvents 类公开的事件比本范例中使用的要多。关于它们的完整列表，参见表 13-1。

表13-1：SystemEvents事件

值	描　　述
DisplaySettingsChanged	用户修改了显示设置
DisplaySettingsChanging	显示设置正在改变
EventsThreadShutdown	侦听系统事件的线程正在终止
InstalledFontsChanged	用户添加或移除了字体
PaletteChanged	用户切换到使用其他调色板的应用程序
PowerModeChanged	用户挂起或恢复了系统
SessionEnded	用户关闭系统或者注销
SessionEnding	用户尝试关闭系统或者注销
SessionSwitch	改变了当前登录用户
TimeChanged	用户更改了系统时间
TimerElapsed	Windows 计时器间隔过期
UserPreferenceChanged	用户更改了系统中的首选项
UserPreferenceChanging	用户正试图修改系统中的首选项

要谨记，这些是系统事件。因此，在处理程序中完成的工作应当尽量少，以使系统能够继续进行下一项任务。

来自 SystemEvents 的通知从专门用于引发这些事件的线程中发出。在 UI 应用程序中，在使用任何信息更新 UI 前，都需要回到正确的用户界面线程中。使用几个方法（Control.BeginInvoke、Control.Inovke 或者 BackgroundWorker）之一来完成此操作。

请注意，在编写本书时，.NET Core（用于跨平台开发的开源版本 .NET）不包括 Microsoft.Win32.SystemEvents 类，因此这个范例将不能工作于 .NET Core，直到有人添加了它为止。

13.1.4 参考

MSDN 文档中的“SystemEvents 类”“PowerModeChangedEventArgs 类”“SessionEndedEventArgs 类”“SessionEndingEventArgs 类”和“SessionSwitchEventArgs 类”主题。

13.2 控制系统服务

13.2.1 问题

你需要通过编程方式操作应用程序与之交互的服务。

13.2.2 解决方案

使用 System.ServiceProcess.ServiceController 类控制服务。ServiceController 允许你与一个已有的服务进行交互并读取和修改其属性。在示例中，它将用于操作 ASP.NET 状态服务。其名称、服务类型和显示名称很容易从 ServiceName、ServiceType 和 DisplayName 属性得到。

```
ServiceController scStateService = new ServiceController("COM+ Event System");
Console.WriteLine($"Service Type: {scStateService.ServiceType.ToString()}");
Console.WriteLine($"Service Name: {scStateService.ServiceName}");
Console.WriteLine($"Display Name: {scStateService.DisplayName}");
```

ServiceType 枚举有一些值，如表 13-2 所示。

表13-2：ServiceType枚举值

值	描　述
Adapter	用于硬件设备的服务
FileSystemDriver	文件系统驱动（内核级别）
InteractiveProcess	与桌面进行通信的服务
KernelDriver	低级别的硬件设备驱动
RecognizerDriver	在启动时用于确定文件系统的驱动
Win32OwnProcess	作为一项服务在其独立进程中运行的 Win32 程序
Win32ShareProcess	作为一项服务在共享进程（如 SvcHost）中运行的 Win32 程序

确定一个服务的依赖性服务是一项非常有用的任务。依赖当前服务的服务可通过 DependentServices 属性进行访问，它是一个 ServiceController 实例数组（每个实例对应一个依赖服务）。

```
foreach (ServiceController sc in scStateService.DependentServices)
    Console.WriteLine($"{scStateService.DisplayName} is depended on by: " +
                        $" {sc.DisplayName}");
```

相比而言，ServicesDependedOn 数组包含当前服务依赖的每一个服务对应的 ServiceController 实例。

```
foreach (ServiceController sc in scStateService.ServicesDependedOn)
    Console.WriteLine(
        $"{scStateService.DisplayName} depends on: {sc.DisplayName}");
```

关于服务最重要的事情之一就是它们所处的状态。如果一个服务被期望运行而没有运行，或者更糟的是期望禁用它（可能因为一个安全风险）而实际没有禁用它，那么它不会起到

很好的作用。要找出当前服务状态，可检查 Status 属性。对此例来说，服务的原始状态将被保存，以便随后从 originalState 变量中恢复它。

```
Console.WriteLine($"Status: {scStateService.Status}");
// 保存原始状态
ServiceControllerStatus originalState = scStateService.Status;
```

既然我们已经建立了合适的访问，就可以开始使用服务的方法。如果服务停止，那么可以使用 Start 方法启动它。首先，检查服务是否停止；然后，一旦在 ServiceController 实例上调用了 Start，调用 WaitForStatus 方法以确保服务启动。WaitForStatus 可以接受一个超时值，使应用程序不必在出现问题时永远等待服务启动。

```
TimeSpan serviceTimeout = TimeSpan.FromSeconds(60);
// 如果服务停止了,启动它
if (scStateService.Status == ServiceControllerStatus.Stopped)
{
    scStateService.Start();
    // 等待服务启动,最多等60秒
    scStateService.WaitForStatus(ServiceControllerStatus.Running,
        serviceTimeout);
}
Console.WriteLine($"Status: {scStateService.Status}");
```

服务也可以暂停。如果服务被暂停，那么应用程序需要通过查看 CanPauseAndContinue 属性检查是否能够继续运行它。如果可以，Continue 方法将再次令服务运行，而 WaitForStatus 方法应当被调用以等待服务运行，代码如下所示。

```
// 如果服务暂停了,恢复运行
if (scStateService.Status == ServiceControllerStatus.Paused)
{
    if (scStateService.CanPauseAndContinue)
    {
        scStateService.Continue();
        // 等待服务启动,最多等60秒
        scStateService.WaitForStatus(ServiceControllerStatus.Running,
            serviceTimeout);
    }
}
Console.WriteLine($"Status: {scStateService.Status}");

// 服务此时应该已在运行
```

确定一种服务能否被停止可通过 CanStop 属性来完成。如果它能被停止，则需要调用 Stop 方法以及 WaitForStatus，代码如下所示。

```
// 是否能够停止服务
if (scStateService.CanStop)
{
    scStateService.Stop();
    // 等待服务停止,最多等60秒
    scStateService.WaitForStatus(ServiceControllerStatus.Stopped,
        serviceTimeout);
}
Console.WriteLine($"Status: {scStateService.Status}");
```

即使 CanStop 可能返回 true，如果我们没有在管理上下文中运行，就会在试图停止服务时得到以下异常。

```
A first chance exception of type 'System.InvalidOperationException' occurred in
System.ServiceProcess.dll
Additional information: Cannot open EventSystem service on computer '.'.
```

请参阅讨论部分中如何为代码设置适当的安全访问权限。

现在是时候将服务设置回当初的状态了。originalState 变量持有原始状态，而 switch 语句包含将服务从当前停止的状态设置为原始状态的动作。

```
// 将服务设置回原始状态
switch (originalState)
{
    case ServiceControllerStatus.Stopped:
        if (scStateService.CanStop)
            scStateService.Stop();
        break;
    case ServiceControllerStatus.Running:
        scStateService.Start();
        // 等待服务启动,最多等60秒
        scStateService.WaitForStatus(ServiceControllerStatus.Running,
            serviceTimeout);
        break;
    case ServiceControllerStatus.Paused:
        // 如果原来是暂停,但现在被停止了,需要先重新启动再暂停
        if (scStateService.Status == ServiceControllerStatus.Stopped)
        {
            scStateService.Start();
            // 等待服务启动,最多等60秒
            scStateService.WaitForStatus(ServiceControllerStatus.Running,
                serviceTimeout);
        }
        // 然后暂停
        if (scStateService.CanPauseAndContinue)
        {
            scStateService.Pause();
            // 等待服务停止,最多等60秒
            scStateService.WaitForStatus(ServiceControllerStatus.Paused,
                serviceTimeout);
        }
        break;
}
```

为了确保服务上的 Status 属性是正确的，应用程序应当在测试 Status 属性值之前调用 Refresh 对其进行更新。一旦应用程序完成对服务的使用，就调用 Close 方法。

```
scStateService.Refresh();
Console.WriteLine($"Status: {scStateService.Status.ToString()}");

//关闭它
scStateService.Close();
```

13.2.3 讨论

现在，服务运行许多操作系统功能。它们通常在某个系统账户（`LocalSystem`、`NetworkService`、`LocalService`）下或者某个已被赋予指定许可和权限的特定用户账户下运行。如果你的应用程序使用一个服务，那么这是在用户程序试图使用它之前，确定用于运行该服务的所有工作是否已建立并正确配置的一个好方法。并不是所有应用程序都直接依赖服务。但如果你的应用程序直接依赖服务，或者你已经编写了一个服务作为应用程序的一部分，那么存在一种简单的方法去检查服务的状态并且纠正可能的情况就很方便了。

当你在操作服务时，就会遇到访问权限问题了。虽然在早期的 Microsoft 操作系统（Windows 7 之前）中你可以调用 `ServiceController` API 而无需任何特定权限，随着用户账户控制（user account control，UAC）的引入，现在必须要在管理上下文中才能访问影响服务运行的方法。你依然无需此访问级别就可以查看服务的属性，但如果想要启动、停止等，就需要提升权限。

要在代码中做到这一点，可以将一个 app.manifest 文件添加到应用程序，通过右键单击项目并选择 Add*New Item（添加新项目），然后选择 Application Manifest File（应用程序清单文件），如图 13-1 所示。

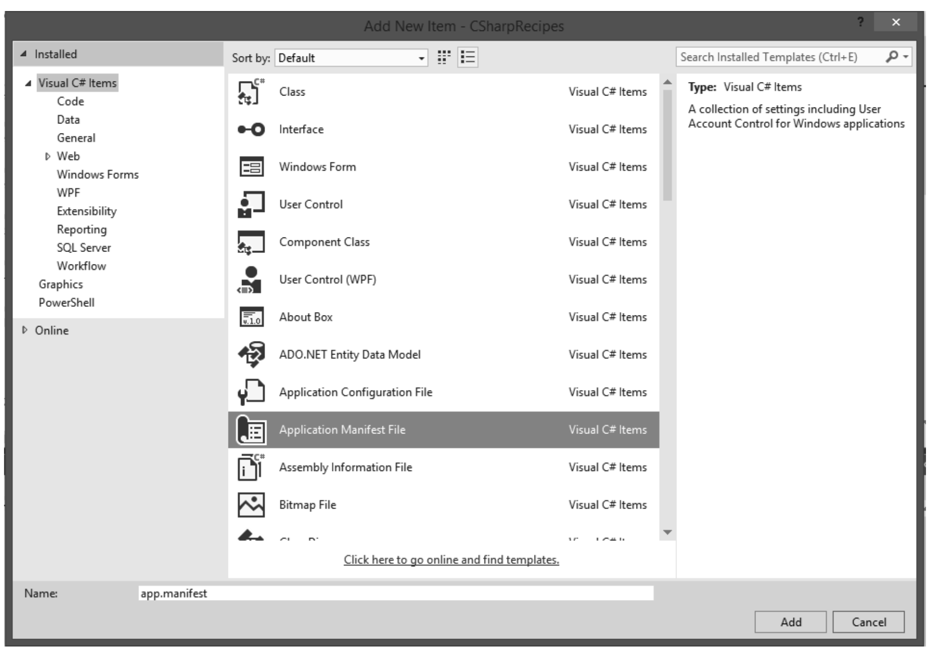


图 13-1：Application Manifest File 创建窗口

在此文件的 asmv1:assembly\trustinfo\security\requestedPrivileges 节，默认请求的执行级别是以调用代码的用户来运行。

```
<requestedExecutionLevel level="asInvoker" uiAccess="false" />
```

要允许访问服务方法，可以通过把 level 属性设置为 requireAdministrator，以使代码需要管理上下文。

```
<!-- 对于范例13.2(即13.2节):控制服务中的服务交互是必需的 -->
<requestedExecutionLevel level="requireAdministrator" uiAccess="false"/>
```

这将确保当代码运行时，它要求用户具有足够的权限来执行我们所请求的行为。

13.2.4 参考

MSDN 文档中的“ServiceControl 类”和“ServiceControllerStatus 枚举”主题。

13.3 列出加载一个程序集的进程

13.3.1 问题

你希望知道当前哪些进程加载了某个给定程序集。

13.3.2 解决方案

使用我们为此目的而创建的 GetProcessesAssemblyIsLoadedIn 方法返回一个加载了给定程序集的进程列表。GetProcessesAssemblyIsLoadedIn 获取程序集的文件名（如 mscoree.dll）以进行查找，然后通过调用 Process.GetProcesses 获得机器上当前运行的进程列表。然后它搜索进程，查看程序集是否加载到它们之中。如果在某一进程中找到程序集，那么就将该 Process 对象投影到一个 Process 对象的可枚举集合。从查询中返回所发现的进程集合的迭代器。

```
public static IEnumerable<Process> GetProcessesAssemblyIsLoadedIn(
    string assemblyFileName)
{
    // System和Idle并不算真正的进程,
    // 所以它们没有关联任何模块,跳过它们

    var processes = from process in Process.GetProcesses()
                    where process.ProcessName != "System" &&
                            process.ProcessName != "Idle"
                    from ProcessModule processModule in process.SafeGetModules()
                    where processModule.ModuleName.Equals(assemblyFileName,
                         StringComparison.OrdinalIgnoreCase)
                    select process;
    return processes;
}
```

Process.GetSafeModules 扩展方法获取调用者被授权可见的模块列表。如果我们只是直接访问 Modules 属性，会得到一系列不同的访问错误，具体取决于调用者的安全上下文。

```
public static ProcessModuleCollection SafeGetModules(this Process process)
{
    List<ProcessModule> listModules = new List<ProcessModule>();
```

```
        ProcessModuleCollection modules =
                new ProcessModuleCollection(listModules.ToArray());

        try
        {
            modules = process.Modules;
        }
        catch (InvalidOperationException) { }
        catch (PlatformNotSupportedException) { }
        catch (NotSupportedException) { }
        catch (Win32Exception wex)
        {
            Console.WriteLine($"Couldn't get modules for {process.ProcessName}: " +
                                $"{wex.Message}");
        }
        // 返回模块集合或者一个空集合
        return modules;
    }
```

13.3.3 讨论

在某些情况下，例如当卸载软件或调试版本冲突时，知道一个程序集是否被加载到多个进程中是很有益的。通过快速获取程序集被加载到的 Process 对象列表，你能够缩小研究的范围。

下列代码使用这一例程查找 .NET 进程。

```
string searchAssm = "mscoree.dll";
var processes = GetProcessesAssemblyIsLoadedIn(searchAssm);
foreach (Process p in processes)
    Console.WriteLine($"Found {searchAssm} in {p.MainModule.ModuleName}");
```

当你运行 GetProcessesAssemblyIsLoadedIn 方法时，用户的安全上下文对于代码能发现多少进程起了很大的作用。如果调用方是不在管理上下文（它必须由用户显式进入）中运行的普通 Windows 用户，你会看到一些进程报告说不能检查它们的模块，如例 13-2 所示。

例 13-2：普通用户安全上下文输出示例

```
Couldn't get modules for dasHost: Access is denied
Couldn't get modules for WUDFHost: Access is denied
Couldn't get modules for StandardCollector.Service: Access is denied
Couldn't get modules for winlogon: Access is denied
Couldn't get modules for svchost: Access is denied
Couldn't get modules for FcsSas: Access is denied
Couldn't get modules for VBCSCompiler: Access is denied
Couldn't get modules for svchost: Access is denied
Couldn't get modules for coherence: Access is denied
Couldn't get modules for coherence: Access is denied
Couldn't get modules for svchost: Access is denied
Couldn't get modules for MOMService: Access is denied
Couldn't get modules for svchost: Access is denied
Couldn't get modules for csrss: Access is denied
Couldn't get modules for svchost: Access is denied
Couldn't get modules for vmms: Access is denied
```

```
Couldn't get modules for dwm: Access is denied
Found mscoree.dll in Microsoft.VsHub.Server.HttpHostx64.exe
Couldn't get modules for wininit: Access is denied
Couldn't get modules for svchost: Access is denied
Couldn't get modules for prl_tools: Access is denied
Couldn't get modules for coherence: Access is denied
Couldn't get modules for MpCmdRun: Access is denied
Couldn't get modules for svchost: Access is denied
Couldn't get modules for svchost: Access is denied
Couldn't get modules for audiodg: Access is denied
Couldn't get modules for mqsvc: Access is denied
Couldn't get modules for WmiApSrv: Access is denied
Couldn't get modules for conhost: Access is denied
Couldn't get modules for sqlwriter: Access is denied
Couldn't get modules for svchost: Access is denied
Couldn't get modules for svchost: Access is denied
Found mscoree.dll in CSharpRecipes.exe
Couldn't get modules for WmiPrvSE: Access is denied
Couldn't get modules for spoolsv: Access is denied
Couldn't get modules for svchost: Access is denied
Couldn't get modules for WmiPrvSE: Access is denied
Couldn't get modules for svchost: Access is denied
Found mscoree.dll in msvsmon.exe
Couldn't get modules for csrss: Access is denied
Couldn't get modules for dllhost: Access is denied
Couldn't get modules for svchost: Access is denied
Couldn't get modules for SearchIndexer: Access is denied
Couldn't get modules for WmiPrvSE: Access is denied
Found mscoree.dll in VBCSCompiler.exe
Couldn't get modules for svchost: Access is denied
Couldn't get modules for OSPPSVC: Access is denied
Couldn't get modules for WmiPrvSE: Access is denied
Couldn't get modules for smss: Access is denied
Couldn't get modules for IpOverUsbSvc: Access is denied
Couldn't get modules for lsass: Access is denied
Couldn't get modules for services: Access is denied
Couldn't get modules for MsMpEng: Access is denied
Couldn't get modules for msdtc: Access is denied
Couldn't get modules for prl_tools_service: Access is denied
Couldn't get modules for inetinfo: Access is denied
Couldn't get modules for sppsvc: Access is denied
```

当我们在管理上下文中运行同样的 `GetProcessesAssemblyIsLoadedIn` 调用时，将得到类似于例 13-3 的输出。

例 13-3：管理用户安全上下文输出示例

```
Found mscoree.dll in VBCSCompiler.exe
Found mscoree.dll in Microsoft.VsHub.Server.HttpHostx64.exe
Found mscoree.dll in msvsmon.exe
Found mscoree.dll in VBCSCompiler.exe
Couldn't get modules for audiodg: Access is denied
Found mscoree.dll in ElevatedPrivilegeActions.vshost.exe
Couldn't get modules for sppsvc: Access is denied
```

因为这是一个诊断函数，你将会需要 `FullTrust` 安全访问来使用该方法。

注意查询跳过了 System 和 Idle 进程的检查，代码如下所示。

```
var processes = from process in Process.GetProcesses()
                where process.ProcessName != "System" &&
                      process.ProcessName != "Idle"
                from ProcessModule processModule in process.SafeGetModules()
                where processModule.ModuleName.Equals(assemblyFileName,
                    StringComparison.OrdinalIgnoreCase)
                select process;
```

Modules 集合不能用于检查这两个进程，因为这样做会引发一个 Win32Exception。对于另外两个进程，你可能也会看到访问被拒绝：audiodg 和 sppsvc。

- audiodg 是一个受 DRM 保护的进程，用于音频驱动程序的宿主，以便这些音频驱动程序能够在与本地登录的用户隔离开的登录会话中运行。
- sppsvc 是一个 Microsoft 软件保护平台服务，用来防止使用未经许可的软件。

由于这两个服务在操作系统中都是很敏感的，你会明白它们为什么无法通过 Process 枚举访问。

13.3.4　参考

MSDN 文档中的“Process 类”“ProcessModule 类”和“GetProcesses 方法”主题。

13.4　使用本地工作站上的消息队列

13.4.1　问题

你需要一种断开应用程序中的两个组件（例如一个 Web 服务端点和处理逻辑）的方法，使得第一个组件只需关心格式化指令。这样，批处理可以发生在第二个组件上。

13.4.2　解决方案

使用消息队列来分隔工作，在第一个和第二个组件中使用此处展示的 MQWorker 类向一个关联的消息队列写入或从中读取消息。

> 消息队列在各方之间提供异步通信协议，意味着消息的发送方和接收方不需要在同一时间与消息队列进行交互。放置到队列上的消息被存储，直到接收方获取它们。消息队列确保消息不会丢失，将工作进行分割，处理不一致的负载，并且通过多个工作者从一个队列中读取来扩展你的应用程序。

MQWorker 使用本地消息排队服务来存储和获取消息。队列路径名在构造函数中提供，并且在 SetUpQueue 方法中检查队列是否存在，代码如下所示。

```
class MQWorker : IDisposable
{
    private bool _disposed;
```

```
        private string _mqPathName;
        MessageQueue _queue;

        public MQWorker(string queuePathName)
        {
            if (string.IsNullOrEmpty(queuePathName))
                throw new ArgumentNullException(nameof(queuePathName));

            _mqPathName = queuePathName;

            SetUpQueue();
        }
```

如果同名队列不存在，SetUpQueue 使用 MessageQueue 类创建一个指定名称的消息队列。它处理消息队列服务在工作站计算机上运行的情况。在此情况下，它使队列私有化，这是工作站上允许的唯一队列类型。

```
    private void SetUpQueue()
    {
        // 检查队列是否存在,不存在就创建它
        if (!MessageQueue.Exists(_mqPathName))
        {
            try
            {
                _queue = MessageQueue.Create(_mqPathName);
            }
            catch (MessageQueueException mqex)
            {
                // 检查是否在一台工作组的计算机上运行
                if (mqex.MessageQueueErrorCode ==
                        MessageQueueErrorCode.UnsupportedOperation)
                {
                    string origPath = _mqPathName;
                    // 工作站模式下必定是一个私有队列
                    int index = _mqPathName.ToLowerInvariant().
                                    IndexOf("private$",
                                        StringComparison.OrdinalIgnoreCase);
                    if (index == -1)
                    {
                        // 获得第一个\
                        index = _mqPathName.IndexOf(@"\",
                                    StringComparison.OrdinalIgnoreCase);
                        // 在每条服务器记录后面添加private$\
                        _mqPathName = _mqPathName.Insert(index + 1, @"private$\");
                        // 重试
                        try
                        {
                            if (!MessageQueue.Exists(_mqPathName))
                                _queue = MessageQueue.Create(_mqPathName);
                            else
                                _queue = new MessageQueue(_mqPathName);
                        }
                        catch (Exception)
                        {
                            // 将原始异常设置为内部(inner)异常
```

```
                            throw new Exception(
                                $"Failed to create message queue with {origPath}" +
                                $" or {_mqPathName}", mqex);
                        }
                    }
                }
            }
        }
        else
        {
            _queue = new MessageQueue(_mqPathName);
        }
    }
```

SendMessage 方法向构造函数中建立的队列发送一条消息。消息体由 body 参数提供，然后创建并填充一个 System.Messaging.Message 的实例。BinaryMessageFormatter 用于格式化消息，因为与默认的 XmlMessageFormatter 相比，它能够使用更少的资源发送更大的消息。通过将 Recoverable 属性设置为 true 可将消息设置为持久的（使消息持久保存直到它们被处理，不会引发机器断电而丢失）。最后设置 Body 并发送消息。

```
    public void SendMessage(string label, string body)
    {
        Message msg = new Message();
        // 标记消息
        msg.Label = label;
        // 用二进制来代替默认的XML格式化
        // 因为它更快(代价是调试时的易读性)
        msg.Formatter = new BinaryMessageFormatter();
        // 将此消息设置为持久的(导致消息将被写入磁盘)
        msg.Recoverable = true;
        msg.Body = body;
        _queue?.Send(msg);
    }
```

ReadMessage 方法通过创建一个 Message 对象并调用其 Receive 方法，从构造函数建立的队列中读取消息。Message 的消息格式化器被设置为 BinaryMessageFormatter，因为它是我们写入队列的方式。最后，从方法中返回消息体。

```
        public string ReadMessage()
        {
            Message msg = null;
            msg = _queue.Receive();
            msg.Formatter = new BinaryMessageFormatter();
            return (string)msg.Body;
        }


        #region IDisposable Members

        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }
```

```
        private void Dispose(bool disposing)
        {
            if (!this._disposed)
            {
                if (disposing)
                    _queue.Dispose();

                _disposed = true;
            }
        }
        #endregion
    }
```

为了展示如何使用 MQWorker 类，下面的示例生成一个 MQWorker。它随后会使用 SendMessage 发送一条消息（一小段 XML）并使用 ReadMessage 获取它。

```
// 注意:必须先设置好消息队列服务,本例才能正常工作
// 可以在"添加/删除Windows组件"中添加此服务

using (MQWorker mqw = new MQWorker(@".\MQWorkerQ")) {
    string xml = "<MyXml><InnerXml location=\"inside\"/></MyXml>";
    Console.WriteLine("Sending message to message queue: " + xml);
    mqw.SendMessage("Label for message", xml);
    string retXml = mqw.ReadMessage();
    Console.WriteLine("Read message from message queue: " + retXml);
}
```

13.4.3 讨论

消息队列在你试图为可扩展性目的而分发处理负载时非常有用。毫无疑问，使用消息队列会为处理增加开销，因为消息必须遍历 MSMQ 的基础架构。尽管如此，一个好处是 MSMQ 允许应用程序在多台机器上运行，因此处理时可能最终获益。另一个好处是消息队列提供可靠的消息异步处理，因此发送方可以确信接收方将会得到消息而无需发送方等待确认。消息队列服务默认未安装，但可以通过控制面板中的“添加 / 删除 Windows 组件”小程序进行安装。

使用消息队列以缓冲大量请求的处理逻辑（例如前面所示的 Web 服务情况）会带来更好的稳定性，在多台机器上使用多个读取进程最终会为应用程序产生更大的吞吐量。

13.4.4 参考

MSDN 文档中的“Message 类”和“MessageQueue 类”主题。

13.5 捕获标准输出流的输出

13.5.1 问题

你希望捕获自己的 C# 程序中将会出现在标准输出流中的输出。

13.5.2 解决方案

使用 Console.SetOut 方法捕获和释放标准输出流。SetOut 将标准输出流设置为传递给它的任何基于 System.IO.TextWriter 的流。为了将输出捕获到一个文件中，可创建一个 StreamWriter 以写入到文件，并且使用 SetOut 设置该写入器。使用 Path.GetTempFileName 获得一个位置，来写入调用代码方可访问的日志。

现在，当调用 Console.WriteLine 时，输出会到达 StreamWriter，而不是 stdout，如下所示。

```
try
{
    Console.WriteLine("Stealing standard output!");
    string logfile = Path.GetTempFileName();
    Console.WriteLine($"Logging to: {logfile}");
    using (StreamWriter writer = new StreamWriter(logfile))
    {
        // 为我们的目的截取stdout
        Console.SetOut(writer);

        Console.WriteLine("Writing to the console... NOT!");

        for (int i = 0; i < 10; i++)
            Console.WriteLine(i);
    }
}
catch(IOException e)
{
    Debug.WriteLine(e.ToString());
    return ;
}
```

为了恢复向标准输出流写入，创建另一个 StreamWriter。这一次调用 Console.OpenStandardOutput 方法获取标准输出流并使用 SetOut 再次设置它。现在调用 Console.WriteLine 将再次出现在控制台上。

```
// 恢复标准输出流,以显示完成消息
StreamWriter standardOutput = new StreamWriter(Console.OpenStandardOutput());
standardOutput.AutoFlush = true;

Console.SetOut(standardOutput);
Console.WriteLine("Back to standard output!");
```

该代码的控制台输出如下所示。

```
Stealing standard output!
Logging to: C:\Users\user\AppData\Local\Temp\tmpFE7C.tmp
Back to standard output!
```

代码执行之后，我们创建的日志文件包含下列内容。

```
Writing to the console... NOT!
0
1
```

```
2
3
4
5
6
7
8
9
```

13.5.3　讨论

在程序内重定向标准输出流看起来有些过时。但是，要考虑到使用其他向该流中写入信息的类的情况。你不希望输出出现在自己的应用程序中，但又必须使用该类。这在创建一个小的启动程序以捕获来自控制台应用程序的输出时或者在使用一个第三方程序集（该程序集不断输出大量可能会让你的用户困扰的详细信息）时也非常有用。

13.5.4　参考

MSDN 文档中的“`Console.SetOut` 方法”“`Console.OpenStandardOutput` 方法”“`Path.GetTempFilePath` 方法”和“`StreamWriter` 类”主题。

13.6　捕获一个进程的标准输出

13.6.1　问题

你需要能够捕获你启动的一个进程的标准输出。

13.6.2　解决方案

使用 `Process.StartInfo` 类的 `RedirectStandardOutput` 属性来捕获进程的输出。通过重定向进程的标准输出流，你可以在进程终止时读取它。`UseShellExecute` 是 `ProcessInfo` 类的一个属性，用于指示运行时是否使用 Windows shell 来启动进程。它默认是打开的（`true`），并且 shell 运行该程序，这意味着不能重定向输出。`UseShellExecute` 需要被关闭（设置为 `false`），以使重定向能够发生。

在此示例中，为 cmd.exe 设置了一个 `Process` 对象，该对象带有用于执行目录列表的参数，然后输出被重定向。创建一个日志文件以保存输出结果，然后调用 `Process.Start` 方法。

```
Process application = new Process();
// 运行命令shell
application.StartInfo.FileName = @"cmd.exe";

// 从当前目录获得目录列表
application.StartInfo.Arguments = @"/Cdir " + Environment.CurrentDirectory;
Console.WriteLine($"Running cmd.exe with arguments:" +
    $"{application.StartInfo.Arguments}");
```

```
// 重定向输出,以便我们能够读取它
application.StartInfo.RedirectStandardOutput = true;
application.StartInfo.UseShellExecute = false;

// 创建一个日志文件以保存结果
string logfile = Path.GetTempFileName();
Console.WriteLine($"Logging to: {logfile}");
using (StreamWriter logger = new StreamWriter(logfile))
{
    // 启动它
    application.Start();
```

一旦进程启动，就可以访问 StandardOutput 流并保存一个引用。应用程序一结束，代码就读取应用程序运行时写入输出流的信息，并将其写入之前建立的日志文件中。最后，关闭日志文件，然后 Process 对象也会被关闭。

```
    application.WaitForExit();

    string output = application.StandardOutput.ReadToEnd();

    logger.Write(output);
}

// 关闭进程对象
application.Close();
```

我们使用 Path.GetTempPathFile 创建的临时日志文件保存的信息类似于下面的输出。

```
Volume in drive C has no label.
Volume Serial Number is DDDD-FFFF

Directory of C:\CS60_Cookbook\CSCB6\CSharpRecipes\bin\Debug

04/11/2015  04:27 PM    <DIR>          .
04/11/2015  04:27 PM    <DIR>          ..
02/05/2015  10:06 PM               724 BigSpenders.xml
02/05/2015  10:05 PM               719 Categories.xml
02/05/2015  04:04 PM            64,566 CSCBCover.bmp
12/31/2014  05:23 PM           489,269 CSharpCookbook.zip
04/11/2015  04:27 PM           495,616 CSharpRecipes.exe
04/11/2015  04:27 PM            31,154 CSharpRecipes.exe.CodeAnalysisLog.xml
02/05/2015  09:53 PM             3,075 CSharpRecipes.exe.config
04/11/2015  04:27 PM                 0
CSharpRecipes.exe.lastcodeanalysissucceeded
04/11/2015  04:27 PM           775,680 CSharpRecipes.pdb
02/05/2015  04:04 PM         5,190,856 EntityFramework.dll
02/05/2015  04:04 PM           620,232 EntityFramework.SqlServer.dll
02/05/2015  04:04 PM           154,645 EntityFramework.SqlServer.xml
02/05/2015  04:04 PM         3,645,119 EntityFramework.xml
03/09/2015  02:51 PM             6,569 IngredientList.txt
04/04/2015  09:55 AM           513,536 Newtonsoft.Json.dll
04/04/2015  09:55 AM           494,336 Newtonsoft.Json.xml
03/09/2015  02:51 PM         4,390,912 Northwind.mdf
04/06/2015  04:11 PM            51,712 NorthwindLinq2Sql.dll
04/06/2015  04:11 PM           128,512 NorthwindLinq2Sql.pdb
```

```
04/11/2015  01:18 PM           573,440 Northwind_log.ldf
03/09/2015  02:51 PM                80 RecipeChapters.txt
04/06/2015  04:11 PM            16,384 SampleClassLibrary.dll
04/06/2015  04:11 PM             1,283 SampleClassLibrary.dll.CodeAnalysisLog.xml
04/06/2015  04:11 PM                 0
SampleClassLibrary.dll.lastcodeanalysissucceeded
04/06/2015  04:11 PM            11,776 SampleClassLibrary.pdb
12/02/2014  03:35 PM               387 SampleClassLibraryTests.xml
04/11/2015  03:48 PM             8,704 SharedCode.dll
04/11/2015  03:48 PM            15,872 SharedCode.pdb
              28 File(s)     17,685,158 bytes
               2 Dir(s)  67,929,718,784 bytes free
```

13.6.3　讨论

重定向标准输出流对于自动构建场景或测试工具等任务是非常有用的。虽然并不像在命令提示符中在一个进程命令行后添加 > 那么简单，但这一方法更加灵活，因为流输出可重新格式化为 XML 或 HTML 以发布到一个网站。它还能提供将数据同时发送到多个位置的机会，而这是 Windows 简单的命令行重定向功能无法做到的。

等待直到应用程序完成之后从再从流中读取，可确保没有死锁问题。如果流在此之前被同步访问，那么有可能父进程会阻塞子进程。至少，子进程将会等待父进程完成从流中读取之后才能继续写入其中。因此，通过把读取延迟到结束之后，避免了子进程的性能降级，而代价是结束之后的一些额外时间。

13.6.4　参考

MSDN 文档中的“`ProcessStartInfo.RedirectStandardOutput` 属性”和“`ProcessStartInfo.UseShellExecute` 属性”主题。

13.7　在它自己的AppDomain中运行代码

13.7.1　问题

你希望运行与应用程序主要部分隔离开的代码。

13.7.2　解决方案

使用 `AppDomain.CreateDomain` 方法创建一个单独的 `AppDomain` 来运行代码。`CreateDomain` 允许应用程序控制创建的 `AppDomain` 的许多方面，例如安全环境、`AppDomain` 设置以及 `AppDomain` 的基本路径。为了举例说明这一点，下列代码创建了 `RunMe` 类的一个实例（在本范例中随后将详细介绍）并调用 `PrintCurrentAppDomainName` 方法。它会输出代码运行所在的 `AppDomain` 的名称。

```
AppDomain myOwnAppDomain = AppDomain.CreateDomain("MyOwnAppDomain");
// 输出当前AppDomain的名称
```

```
RunMe rm = new RunMe();
rm.PrintCurrentAppDomainName();
```

现在，通过调用 AppDomain 上的 CreateInstance 在 "MyOwnAppDomain" AppDomain 中生成 RunMe 类的一个实例。我们将用于构造该类型所必需的模块和类型信息传递给 CreateInstance，而它返回一个 ObjectHandle。

然后，通过返回的 ObjectHandle 并使用 Unwrap 方法将其转换为一个 RunMe 引用，获得一个在 AppDomain 中运行的实例的代理。

```
// 在新的应用程序域中创建RunMe
Type adType = typeof(RunMe);
ObjectHandle objHdl =
    myOwnAppDomain.CreateInstance(adType.Module.Assembly.FullName,
        adType.FullName);
// 拆开引用
RunMe adRunMe = (RunMe)objHdl.Unwrap();
```

在 "MyOwnAppDomain" AppDomain 中的 RunMe 实例上调用 PrintCurrentAppDomainName 方法，它会输出 "Hello from MyOwnAppDomain!"。通过 AppDomain.Unload 卸载 AppDomain，然后程序终止。

```
// 对工具箱进行一次调用
adRunMe.PrintCurrentAppDomainName();

// 现在卸载应用程序域
AppDomain.Unload(myOwnAppDomain);
```

RunMe 类的定义如下所示。它继承自 MarshalByRefObject，因为它允许你在调用 ObjectHandle 上的 Unwrap 时获取代理，并将类上的调用远程传递到新的 AppDomain。PrintCurrentAppDomainName 方法简单地访问当前 AppDomain 上的 FriendlyName 属性并输出“Hello from {*AppDomain*}!”消息。

```
public class RunMe : MarshalByRefObject
{
    public RunMe()
    {
        PrintCurrentAppDomainName();
    }

    public void PrintCurrentAppDomainName()
    {
        string name = AppDomain.CurrentDomain.FriendlyName;
        Console.WriteLine($"Hello from {name}!");
    }
}
```

该示例的输出如下所示。

```
Hello from CSharpRecipes.exe!
Hello from CSharpRecipes.exe!
Hello from MyOwnAppDomain!
Hello from MyOwnAppDomain!
```

13.7.3 讨论

将代码隔离到一个单独的 AppDomain 中对于像本示例这样琐碎的情况有点浪费，但是它举例说明了可以在应用程序创建的 AppDomain 中远程执行代码。CreateDomain 方法有 6 种重载，每种都为 AppDomain 的创建增加了一点复杂性。在隔离或定制配置的好处超过了设置一个单独 AppDomain 以及调试其中代码的复杂性的情况下，它是一种很有用的工具。一个很好的真实示例就是建立一个单独的 AppDomain 以运行通常的 ASP.NET 环境之外的 ASP.NET 网页（虽然这的确是一个很重要的用法）或者是将第三方代码加载到第二个 AppDomain 中进行隔离。

13.7.4 参考

MSDN 文档中的“AppDomain 类”“AppDomain.CreateDomain 方法”和“ObjectHandle 类”主题。

13.8 确定当前操作系统的操作系统和Service Pack版本

13.8.1 问题

你希望知道当前操作系统和 Service Pack 的信息。

13.8.2 解决方案

使用例 13-4 所示的 GetOSAndServicePack 方法，获得代表当前操作系统和 Service Pack 的字符串。GetOSAndServicePack 使用 Environment.OSVersion 属性获得操作系统（OS）的版本信息，然后从中确定 OS 的“官方”名称。从 Environment.OSVersion 中获取的 OperatingSystem 类拥有一个名为 ServicePack 的 Service Pack 属性。这些值全都是作为操作系统名称、版本和 Service Pack 字符串返回的。

例 13-4：GetOSAndServicePack 方法

```
public static string GetOSAndServicePack()
{
    // 获得当前OS信息
    OperatingSystem os = Environment.OSVersion;
    RegistryKey rk =
        Registry.LocalMachine.OpenSubKey(
            @"SOFTWARE\Microsoft\Windows NT\CurrentVersion");
    string osText = (string)rk?.GetValue("ProductName");
    if (string.IsNullOrWhiteSpace(osText))
        osText = os.VersionString;
    else
        osText = (
            $"{osText} {os.Version.Major}.{os.Version.Minor}.{os.Version.Build}");
    if (!string.IsNullOrWhiteSpace(os.ServicePack))
```

```
        osText = $"{osText} {os.ServicePack}";
    return osText;
}
```

13.8.3 讨论

让你的应用程序能够获知当前操作系统和 Service Pack 允许你在调试报告和应用程序的“关于”对话框（如果有的话）中包含该信息。这一简单知识通过支持部门传送后，能够节省你数小时的调试时间。让这一信息可用是很值得的，这样用户支持部门就能够在客户无法定位时轻松指导他们。

13.8.4 参考

MSDN 文档中的“`Environment.OSVersion` 属性”和“`OperatingSystem` 类”主题。

关于作者

Jay Hilyard 为 Windows 平台开发应用已经有 20 多年，为 .NET 平台开发应用也超过了 15 年。他在 *MSDN Magazine* 上发表过很多文章，目前在新罕布什尔州朴茨茅斯的 Newmarket（Amadeus 的一家子公司）工作。

Stephen Teilhet 从 pre-alpha 版就开始使用 .NET 平台，并且一直使用至今。他任职于 IBM，是源代码静态安全分析工具的主管安全研究员。这一工具用于发现多种语言中的安全漏洞，例如 C# 和 Visual Basic。

关于封面

《C# 经典实例（第 4 版）》封面上的动物是袜带蛇（Thamnophis sirtalis）。之所以这么命名，是因为它们身体上的纵向条纹看起来十分像是过去用于固定男式短袜的袜带。袜带蛇很容易通过其独特的条纹辨识出来：在背部的中间部分有窄条纹，而在两侧是宽条纹。颜色和样式变化使它们能够融入所处的自然环境中，帮助它们躲避捕食者。它们是北美最常见的一种蛇，并且是在阿拉斯加发现的唯一蛇类。

袜带蛇有着龙骨状鳞片，沿鳞片中轴向下有一条或多条叉，给它们一种粗糙的纹理和暗淡的外观。成年袜带蛇身长一般在 46~130 厘米之间（一英尺半到四英尺）。雌蛇通常比雄蛇大，但尾巴较短，身体和尾巴相连处有隆起。

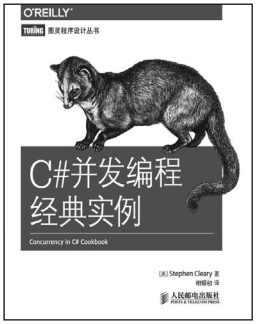
雌袜带蛇是卵胎生动物，意味着小蛇是从柔软的蛋中孕育出生的。雌蛇分娩时，大部分的蛋壳和粘膜都已经破开，所以小蛇会直接降生。偶尔，小蛇出生时还在软壳里。雌蛇通常会生 10 到 40 条小蛇；一条袜带蛇生出的存活小蛇数量最高记录是 98 条。一旦小蛇从母体脱离，它们就完全独立并且需要保护自己。在这段时间，它们是最容易遭到捕食的，超过一半的小蛇都会在 1 岁前死掉。

只有少数动物能捕食蟾蜍、蝾螈等具有很强化学防卫能力的两栖动物，袜带蛇就是其中之一。尽管食物取决于它们所处的环境，但袜带蛇主要还是吃蚯蚓和两栖动物，偶尔会吃雏鸟、鱼和小型啮齿目动物。袜带蛇有毒液（但对人体无害），用于使猎物昏迷或者死亡，然后将其整个吞下。

很多 O'Reilly 封面上的动物都濒临灭绝，它们对于地球都是非常重要的。想要了解更多有关如何帮助这些动物的信息，请访问 http://animals.oreilly.com。

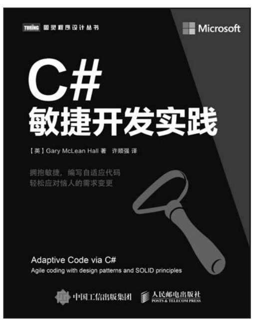
封面图片来自多佛画报档案中一幅 19 世纪的版画。

延 展 阅 读



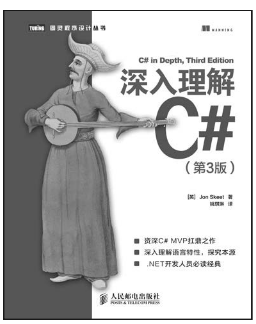
本书全面讲解C#并发编程技术，侧重于.NET平台上较新、较实用的方法。全书分为几大部分：首先介绍几种并发编程技术，包括异步编程、并行编程、TPL数据流、响应式编程；然后阐述一些重要的知识点，包括测试技巧、互操作、取消并发、函数式编程与OOP、同步、调度；最后介绍了几个实用技巧。全书共包含75个有配套源码的实用方法，可用于服务器程序、桌面程序和移动端应用的开发。

书号：978-7-115-37427-1
定价：49.00 元



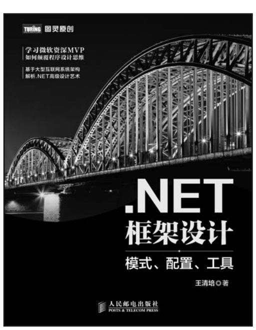
本书共分为敏捷基础、编写SOLID代码和自适应实例三大部分，将理论与实践相结合，介绍了当前使用Microsoft .NET Framework进行C#编程的最佳实践，详尽探讨了C#开发人员如何应用Scrum等敏捷方案实现高质量、自适应的代码，并给出大量代码示例，是.NET中高级程序员进阶的实用指南。

书号：978-7-115-42789-2
定价：69.00 元



本书是世界顶级技术专家“十年磨一剑”的经典之作，在C#和.NET领域享有盛誉。与其他泛泛介绍C#的书籍不同，本书深度探究C#的特性，并结合技术发展，引领读者深入C#的时空。作者从语言设计的动机出发，介绍支持这些特性的核心概念。作者将新的语言特性放在C#语言发展的背景之上，用极富实际意义的示例，向读者展示编写代码和设计解决方案的最佳方式。

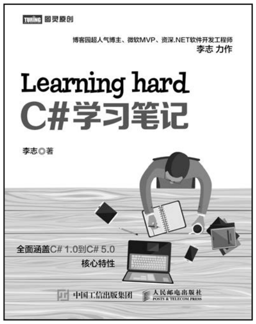
书号：978-7-115-34642-1
定价：99.00 元



本书总结了框架设计的整体思路和经验，包含了常见应用框架设计的模式、框架灵活性的配置和框架工具的支持，有助于读者了解框架设计的核心思想，加深对框架设计的理解，快速掌握框架设计的技巧，并在研究其他框架时能够做到举一反三。

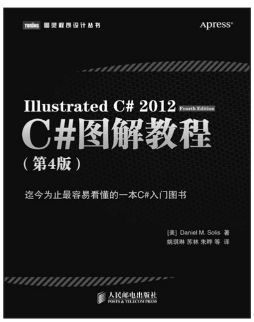
书号：978-7-115-38028-9
定价：49.00 元

延 展 阅 读



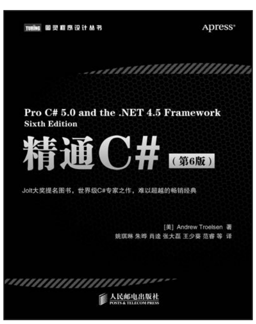
本书是一本面向C#初学者的实用教程,由浅入深地讲解了C#的基础语法和重要特性，分析了在开发中必须掌握的技术要领和经验心得。语言浅显易懂、轻松幽默，通过精心选择的实例和详尽的代码全面介绍了C#最具特色的关键知识点，有助于初学者迅速从一个C#开发的门外汉成长为全面掌握技术要领的开发人员。

书号：978-7-115-38292-4
定价：49.00 元



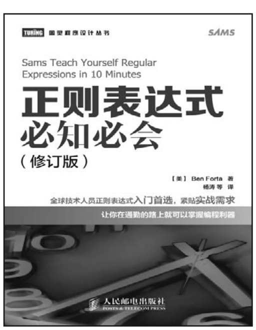
本书是广受赞誉C#图解教程的最新版本。作者在本书中创造了一种全新的可视化叙述方式，以图文并茂的形式、朴实简洁的文字，并辅之以大量表格和代码示例，全面、直观地阐述了C#语言的各种特性。通过本书，读者能够快速、深入地理解C#，为自己的编程生涯打下良好的基础。

书号：978-7-115-32090-2
定价：89.00 元



本书是C# 领域久负盛名的经典著作，深入全面地叙述了C#编程语言和.NET平台的核心内容，并以大量示例剖析相关概念。全书分为八部分内容：C#和.NET平台、C#核心编程结构、C#面向对象编程、高级C#编程结构、用.NET程序集编程、.NET基础类库、WPF和ASP.NET Web Forms。

书号：978-7-115-32181-7
定价：159.00 元



正则表达式是一种威力无比强大的武器，几乎在所有的程序设计语言里和计算机平台上都可以用它来完成各种复杂的文本处理工作。本书从简单的文本匹配开始，循序渐进地介绍了很多复杂内容，其中包括回溯引用、条件性求值和前后查找，等等。每章都为读者准备了许多简明又实用的示例，有助于全面、系统、快速掌握正则表达式，并运用它们去解决实际问题。

书号：978-7-115-37799-9
定价：29.00 元

O'REILLY®

C#经典实例（第4版）

这是一本全面的C#编程参考书，用150多个范例详细探讨了C#开发中的诸多问题。所有范例中的代码均经过验证，可以直接在应用程序中重用。

第4版重新编写了许多解决方案，以充分利用C#最近的创新，例如新的表达式级别功能、成员声明功能和语句级别功能。本书还在范例中纳入了动态编程和异步编程的新应用，帮助读者了解如何应用这些语言特性。

本书涵盖以下主题：

- 类和泛型
- 集合、枚举器和迭代器
- 数据类型
- LINQ和lambda表达式
- 异常处理
- 反射和动态编程
- 正则表达式
- 文件系统交互
- 网络和Web
- XML的使用
- 线程、同步和并发

"一本出色的编程指南，适合随时放在手边参考。书中的解决方案和小提示可以帮助开发人员节省大量时间。"

——Steve Munyan
国际权威评级机构晨星旗下
ByAllAccounts公司
高级软件工程师经理

Jay Hilyard拥有20多年为Windows平台开发应用程序的经验，为.NET平台开发应用也超过了15年。他在*MSDN Magazine*上发表过很多文章，目前在新罕布什尔州朴茨茅斯的Newmarket（Amadeus的一家子公司）工作。

Stephen Teilhet从pre-alpha版就开始使用.NET平台，并且一直使用至今。他任职于IBM，是源代码静态安全分析工具的主管安全研究员。这一工具用于发现多种语言中的安全漏洞，如C#和Visual Basic。

.NET

封面设计：Ellie Volckhausen 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机 / 程序设计 / C#

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

ISBN 978-7-115-43509-5

ISBN 978-7-115-43509-5
定价：129.00元